22783 U.S. PTO
091605

PTO/SB/16 (09-04)
Approved for use through 07/31/2006. OMB 0651-0032
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

**PROVISIONAL APPLICATION FOR PATENT COVER SHEET**

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53(c).

**Express Mail Label No.** EV615580501US

11296 U.S. PTO
60718185
091605

**INVENTOR(S)**

| Given Name (first and middle [if any]) | Family Name or Surname | Residence (City and either State or Foreign Country) |
|---|---|---|
| Rick | Orsini | Flower Mound, TX 75028 |
| Mark | O'Hare | Coto De Caza, CA 92679 |
| Roger | Davenport | Campbell, TX 75422 |
| Paras | Traganos | Lexington, MA 02421 |
| Steve | Winick | Roslyn Heights, NY 11577 |

*Additional inventors are being named on the _____ separately numbered sheets attached hereto*

**TITLE OF THE INVENTION (500 characters max)**

SECURE DATA PARSER METHOD AND SYSTEM

*Direct all correspondence to:* **CORRESPONDENCE ADDRESS**

☒ The address corresponding to Customer Number: 1473

*OR*

☐ Firm or Individual Name

Address

| City | State | Zip |
|---|---|---|
| Country | Telephone | Fax |

**ENCLOSED APPLICATION PARTS *(check all that apply)***

☒ Specification *Number of Pages* 333
☐ CD(s), Number _____
☒ Drawing(s) *Number of Sheets* 36
☐ Other (specify) _____
☒ Application Data Sheet. See 37 CFR 1.76

**METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT**

☒ Applicant claims small entity status. See 37 CFR 1.27.

FILING FEE Amount ($): 850.00

☐ A check or money order is enclosed to cover the filing fees.

☐ Payment by credit card. Form PTO-2038 is attached.

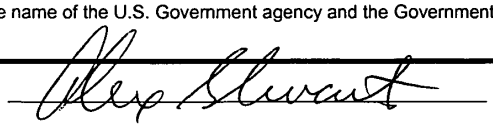
☒ The Director is hereby authorized to charge filing fees or credit any overpayment to Deposit Account Number: 06-1075 Order No. 003960-004

A duplicative copy of this form is enclosed for fee processing.

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

☒ No.

☐ Yes, the name of the U.S. Government agency and the Government contract number are: _____

SIGNATURE 

Date September 16, 2005

TYPED or PRINTED NAME Alexander Shvarts

REGISTRATION NO *(if appropriate)* 47,943

TELEPHONE 212-596-9000

Docket Number: SEC-3 PROV

***USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT***

This collection of information is required by 37 CFR 1.51. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.11 and 1.14. This collection is estimated to take 8 hours to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, VA 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. **SEND TO: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.**

PATENTS
Attorney Docket No. SEC-3 Prov.

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

PROVISIONAL PATENT APPLICATION

Applicants : Rick Orsini, et al.

For : SECURE DATA PARSER METHOD AND SYSTEM

Mail Stop Provisional Application
Hon. Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

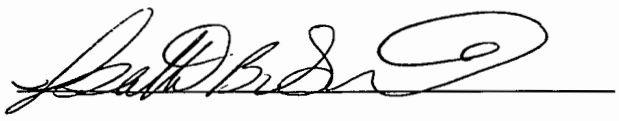
"EXPRESS MAIL" mailing label number EV615580501US

Date of Deposit: September 16, 2005

I hereby certify that this certification and the following papers and fees:

1. Provisional Application Cover Sheet (in duplicate);
2. Specification (333 pages);
3. Drawings (36 sheets);
4. Print EFS; and
5. Postcard receipt.

are being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 C.F.R. § 1.10 on the date indicated above and are addressed to Mail Stop PROVISIONAL PATENT APPLICATION, Hon. Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

Name: 

Express Mail Label No.;
EV615580501US



# SECURE DATA PARSER METHOD AND SYSTEM

## Field of the Invention

**[0001]** The present invention relates in general to a system for securing data from unauthorized access or use.

## Background of the Invention

**[0002]** In today's society, individuals and businesses conduct an ever-increasing amount of activities on and over computer systems. These computer systems, including proprietary and non-proprietary computer networks, are often storing, archiving, and transmitting all types of sensitive information. Thus, an ever-increasing need exists for ensuring data stored and transmitted over these systems cannot be read or otherwise compromised.

**[0003]** One common solution for securing computer systems is to provide login and password functionality. However, password management has proven to be quite costly with a large percentage of help desk calls relating to password issues. Moreover, passwords provide little security in that they are generally stored in a file susceptible to inappropriate access, through, for example, brute-force attacks.

**[0004]** Another solution for securing computer systems is to provide cryptographic infrastructures. Cryptography, in general, refers to protecting data by transforming, or encrypting, it into an unreadable format. Only those who possess the key(s) to the encryption can decrypt the data into a useable format. Cryptography is used to identify users, e.g., authentication, to allow access privileges, e.g., authorization, to create digital certificates and signatures, and the like. One popular cryptography system is a public key system that uses two keys, a public key known to everyone and a private key known only to the individual or business owner thereof. Generally, the data encrypted with one key is decrypted with the other and neither key is recreatable from the other.

**[0005]** Unfortunately, even the foregoing typical public-key cryptographic systems are still highly reliant on the user for security. For example, cryptographic systems issue the private key to the user, for example, through the user's browser. Unsophisticated users then generally store the private key on a hard drive accessible to others through an open computer system, such as, for example, the Internet. On the other hand, users may choose poor names for files containing their private key, such as, for example, "key." The result of the foregoing and other acts is to allow the key or keys to be susceptible to compromise.

**[0006]** In addition to the foregoing compromises, a user may save his or her private key on a computer system configured with an archiving or backup system,

potentially resulting in copies of the private key traveling through multiple computer storage devices or other systems. This security breach is often referred to as "key migration." Similar to key migration, many applications provide access to a user's private key through, at most, simple login and password access. As mentioned in the foregoing, login and password access often does not provide adequate security.

**[0007]** One solution for increasing the security of the foregoing cryptographic systems is to include biometrics as part of the authentication or authorization. Biometrics generally include measurable physical characteristics, such as, for example, finger prints or speech that can be checked by an automated system, such as, for example, pattern matching or recognition of finger print patterns or speech patterns. In such systems, a user's biometric and/or keys may be stored on mobile computing devices, such as, for example, a smartcard, laptop, personal digital assistant, or mobile phone, thereby allowing the biometric or keys to be usable in a mobile environment.

**[0008]** The foregoing mobile biometric cryptographic system still suffers from a variety of drawbacks. For example, the mobile user may lose or break the smartcard or portable computing device, thereby having his or her access to potentially important data entirely cut-off. Alternatively, a malicious person may steal the mobile user's smartcard or portable computing device and use it to effectively steal the mobile user's digital credentials. On the other hand, the portable-computing

device may be connected to an open system, such as the Internet, and, like passwords, the file where the biometric is stored may be susceptible to compromise through user inattentiveness to security or malicious intruders.

Summary of the Invention

**[0009]** Based on the foregoing, a need exists to provide a cryptographic system whose security is user-independent while still supporting mobile users.

**[0010]** Accordingly, one aspect of the present invention is to provide a method for securing virtually any type of data from unauthorized access or use. The method comprises one or more steps of parsing, splitting and/or separating the data to be secured into two or more parts or portions. The method also comprises encrypting the data to be secured. Encryption of the data may be performed prior to or after the first parsing, splitting and/or separating of the data. In addition, the encrypting step may be repeated for one or more portions of the data. Similarly, the parsing, splitting and/or separating steps may be repeated for one or more portions of the data. The method also optionally comprises storing the parsed, split and/or separated data that has been encrypted in one location or in multiple locations. This method also optionally comprises reconstituting or re-assembling the secured data into its original form for authorized access or use. This method may be incorporated into the operations of any computer, server,

engine or the like, that is capable of executing the desired steps of the method.

**[0011]** Another aspect of the present invention provides a system for securing virtually any type of data from unauthorized access or use. This system comprises a data splitting module, a cryptographic handling module, and, optionally, a data assembly module. The system may, in one embodiment, further comprise one or more data storage facilities where secure data may be stored.

**[0012]** Accordingly, one aspect of the invention is to provide a secure server, or trust engine, having server-centric keys, or in other words, storing cryptographic keys and user authentication data on a server. According to this embodiment, a user accesses the trust engine in order to perform authentication and cryptographic functions, such as, but not limited to, for example, authentication, authorization, digital signing and generation, storage, and retrieval of certificates, encryption, notary-like and power-of-attorney-like actions, and the like.

**[0013]** Another aspect of the invention is to provide a reliable, or trusted, authentication process. Moreover, subsequent to a trustworthy positive authentication, a wide number of differing actions may be taken, from providing cryptographic technology, to system or device authorization and access, to permitting use or control of one or a wide number of electronic devices.

**[0014]** Another aspect of the invention is to provide cryptographic keys and authentication data in an environment where they are not lost, stolen, or

compromised, thereby advantageously avoiding a need to continually reissue and manage new keys and authentication data. According to another aspect of the invention, the trust engine allows a user to use one key pair for multiple activities, vendors, and/or authentication requests. According to yet another aspect of the invention, the trust engine performs at least one step of cryptographic processing, such as, but not limited to, encrypting, authenticating, or signing, on the server side, thereby allowing clients or users to possess only minimal computing resources.

**[0015]** According to yet another aspect of the invention, the trust engine includes one or multiple depositories for storing portions of each cryptographic key and authentication data. The portions are created through a data splitting process that prohibits reconstruction without a predetermined portion from more than one location in one depository or from multiple depositories. According to another embodiment, the multiple depositories may be geographically remote such that a rogue employee or otherwise compromised system at one depository will not provide access to a user's key or authentication data.

**[0016]** According to yet another embodiment, the authentication process advantageously allows the trust engine to process multiple authentication activities in parallel. According to yet another embodiment, the trust engine may advantageously track failed access attempts and thereby limit the number of times malicious intruders may attempt to subvert the system.

[0017] According to yet another embodiment, the trust engine may include multiple instantiations where each trust engine may predict and share processing loads with the others. According to yet another embodiment, the trust engine may include a redundancy module for polling a plurality of authentication results to ensure that more than one system authenticates the user.

[0018] Therefore, one aspect of the invention includes a secure cryptographic system, which may be remotely accessible, for storing data of any type, including, but not limited to, a plurality of private cryptographic keys to be associated with a plurality of users. The cryptographic system associates each of the plurality of users with one or more different keys from the plurality of private cryptographic keys and performs cryptographic functions for each user using the associated one or more different keys without releasing the plurality of private cryptographic keys to the users. The cryptographic system comprises a depository system having at least one server which stores the data to be secured, such as a plurality of private cryptographic keys and a plurality of enrollment authentication data. Each enrollment authentication data identifies one of multiple users and each of the multiple users is associated with one or more different keys from the plurality of private cryptographic keys. The cryptographic system also may comprise an authentication engine which compares authentication data received by one of the multiple users to enrollment authentication data corresponding to the one of multiple users and received from the depository

system, thereby producing an authentication result. The cryptographic system also may comprise a cryptographic engine which, when the authentication result indicates proper identification of the one of the multiple users, performs cryptographic functions on behalf of the one of the multiple users using the associated one or more different keys received from the depository system. The cryptographic system also may comprise a transaction engine connected to route data from the multiple users to the depository server system, the authentication engine, and the cryptographic engine.

**[0019]** Another aspect of the invention includes a secure cryptographic system that is optionally remotely accessible. The cryptographic system comprises a depository system having at least one server which stores at least one private key and any other data, such as, but not limited to, a plurality of enrollment authentication data, wherein each enrollment authentication data identifies one of possibly multiple users. The cryptographic system may also optionally comprise an authentication engine which compares authentication data received by users to enrollment authentication data corresponding to the user and received from the depository system, thereby producing an authentication result. The cryptographic system also comprises a cryptographic engine which, when the authentication result indicates proper identification of the user, performs cryptographic functions on behalf of the user using at least said private key, which may be received from the depository system. The cryptographic system may

also optionally comprise a transaction engine connected to route data from the users to other engines or systems such as, but not limited to, the depository server system, the authentication engine, and the cryptographic engine.

**[0020]** Another aspect of the invention includes a method of facilitating cryptographic functions. The method comprises associating a user from multiple users with one or more keys from a plurality of private cryptographic keys stored on a secure location, such as a secure server. The method also comprises receiving authentication data from the user, and comparing the authentication data to authentication data corresponding to the user, thereby verifying the identity of the user. The method also comprises utilizing the one or more keys to perform cryptographic functions without releasing the one or more keys to the user.

**[0021]** Another aspect of the invention includes an authentication system for uniquely identifying a user through secure storage of the user's enrollment authentication data. The authentication system comprises one or more data storage facilities, wherein each data storage facility includes a computer accessible storage medium which stores at least one of portions of enrollment authentication data. The authentication system also comprises an authentication engine which communicates with the data storage facility or facilities. The authentication engine comprises a data splitting module which operates on the enrollment authentication data to create portions, a data assembling

module which processes the portions from at least one of the data storage facilities to assemble the enrollment authentication data, and a data comparator module which receives current authentication data from a user and compares the current authentication data with the assembled enrollment authentication data to determine whether the user has been uniquely identified.

**[0022]** Another aspect of the invention includes a cryptographic system. The cryptographic system comprises one or more data storage facilities, wherein each data storage facility includes a computer accessible storage medium which stores at least one portion of one ore more cryptographic keys. The cryptographic system also comprises a cryptographic engine which communicates with the data storage facilities. The cryptographic engine also comprises a data splitting module which operate on the cryptographic keys to create portions, a data assembling module which processes the portions from at least one of the data storage facilities to assemble the cryptographic keys, and a cryptographic handling module which receives the assembled cryptographic keys and performs cryptographic functions therewith.

**[0023]** Another aspect of the invention includes a method of storing any type of data, including, but not limited to, authentication data in geographically remote secure data storage facilities thereby protecting the data against composition of any individual data storage facility. The method comprises receiving data at a trust engine, combining at the trust engine the data with a first substantially random value to form a first combined

value, and combining the data with a second substantially random value to form a second combined value. The method comprises creating a first pairing of the first substantially random value with the second combined value, creating a second pairing of the first substantially random value with the second substantially random value, and storing the first pairing in a first secure data storage facility. The method comprises storing the second pairing in a second secure data storage facility remote from the first secure data storage facility.

**[0024]** Another aspect of the invention includes a method of storing any type of data, including, but not limited to, authentication data comprising receiving data, combining the data with a first set of bits to form a second set of bits, and combining the data with a third set of bits to form a fourth set of bits. The method also comprises creating a first pairing of the first set of bits with the third set of bits. The method also comprises creating a second pairing of the first set of bits with the fourth set of bits, and storing one of the first and second pairings in a first computer accessible storage medium. The method also comprises storing the other of the first and second pairings in a second computer accessible storage medium.

**[0025]** Another aspect of the invention includes a method of storing cryptographic data in geographically remote secure data storage facilities thereby protecting the cryptographic data against comprise of any individual data storage facility. The method comprises receiving

cryptographic data at a trust engine, combining at the trust engine the cryptographic data with a first substantially random value to form a first combined value, and combining the cryptographic data with a second substantially random value to form a second combined value. The method also comprises creating a first pairing of the first substantially random value with the second combined value, creating a second pairing of the first substantially random value with the second substantially random value, and storing the first pairing in a first secure data storage facility. The method also comprises storing the second pairing in a secure second data storage facility remote from the first secure data storage facility.

[0026] Another aspect of the invention includes a method of storing cryptographic data comprising receiving authentication data and combining the cryptographic data with a first set of bits to form a second set of bits. The method also comprises combining the cryptographic data with a third set of bits to form a fourth set of bits, creating a first pairing of the first set of bits with the third set of bits, and creating a second pairing of the first set of bits with the fourth set of bits. The method also comprises storing one of the first and second pairings in a first computer accessible storage medium, and storing the other of the first and second pairings in a second computer accessible storage medium.

[0027] Another aspect of the invention includes a method of handling sensitive data of any type or form in a cryptographic system, wherein the sensitive data exists

in a useable form only during actions by authorized users, employing the sensitive data. The method also comprises receiving in a software module, substantially randomized or encrypted sensitive data from a first computer accessible storage medium, and receiving in the software module, substantially randomized or encrypted data which may or may not be sensitive data, from one or more other computer accessible storage medium. The method also comprises processing the substantially randomized pre-encrypted sensitive data and the substantially randomized or encrypted data which may or may not be sensitive data, in the software module to assemble the sensitive data and employing the sensitive data in a software engine to perform an action. The action includes, but is not limited to, one of authenticating a user and performing a cryptographic function.

**[0028]** Another aspect of the invention includes a secure authentication system. The secure authentication system comprises a plurality of authentication engines. Each authentication engine receives enrollment authentication data designed to uniquely identify a user to a degree of certainty. Each authentication engine receives current authentication data to compare to the enrollment authentication data, and each authentication engine determines an authentication result. The secure authentication system also comprises a redundancy system which receives the authentication result of at least two of the authentication engines and determines whether the user has been uniquely identified.

Brief Description of the Drawings

[0029] The present invention is described in more detail below in connection with the attached drawings, which are meant to illustrate and not to limit the invention, and in which:

[0030] FIGURE 1 illustrates a block diagram of a cryptographic system, according to aspects of an embodiment of the invention;

[0031] FIGURE 2 illustrates a block diagram of the trust engine of FIGURE 1, according to aspects of an embodiment of the invention;

[0032] FIGURE 3 illustrates a block diagram of the transaction engine of FIGURE 2, according to aspects of an embodiment of the invention;

[0033] FIGURE 4 illustrates a block diagram of the depository of FIGURE 2, according to aspects of an embodiment of the invention;

[0034] FIGURE 5 illustrates a block diagram of the authentication engine of FIGURE 2, according to aspects of an embodiment of the invention;

[0035] FIGURE 6 illustrates a block diagram of the cryptographic engine of FIGURE 2, according to aspects of an embodiment of the invention;

[0036] FIGURE 7 illustrates a block diagram of a depository system, according to aspects of another embodiment of the invention;

[0037] FIGURE 8 illustrates a flow chart of a data splitting process according to aspects of an embodiment of the invention;

[0038] FIGURE 9, Panel A illustrates a data flow of an enrollment process according to aspects of an embodiment of the invention;

[0039] FIGURE 9, Panel B illustrates a flow chart of an interoperability process according to aspects of an embodiment of the invention;

[0040] FIGURE 10 illustrates a data flow of an authentication process according to aspects of an embodiment of the invention;

[0041] FIGURE 11 illustrates a data flow of a signing process according to aspects of an embodiment of the invention;

[0042] FIGURE 12 illustrates a data flow and an encryption/decryption process according to aspects and yet another embodiment of the invention;

[0043] FIGURE 13 illustrates a simplified block diagram of a trust engine system according to aspects of another embodiment of the invention;

[0044] FIGURE 14 illustrates a simplified block diagram of a trust engine system according to aspects of another embodiment of the invention;

[0045] FIGURE 15 illustrates a block diagram of the redundancy module of FIGURE 14, according to aspects of an embodiment of the invention;

[0046] FIGURE 16 illustrates a process for evaluating authentications according to one aspect of the invention;

[0047] FIGURE 17 illustrates a process for assigning a value to an authentication according to one aspect as shown in FIGURE 16 of the invention;

[0048] FIGURE 18 illustrates a process for performing trust arbitrage in an aspect of the invention as shown in FIGURE 17; and

[0049] FIGURE 19 illustrates a sample transaction between a user and a vendor according to aspects of an embodiment of the invention where an initial web based contact leads to a sales contract signed by both parties.

[0050] FIGURE 20 illustrates a sample user system with a cryptographic service provider module which provides security functions to a user system.

[0051] FIGURE 21 illustrates a process for parsing, splitting and/or separating data with encryption and storage of the encryption master key with the data.

[0052] FIGURE 22 illustrates a process for parsing, splitting and/or separating data with encryption and storing the encryption master key separately from the data.

[0053] FIGURE 23 illustrates the intermediary key process for parsing, splitting and/or separating data with encryption and storage of the encryption master key with the data.

[0054] FIGURE 24 illustrates the intermediary key process for parsing, splitting and/or separating data with encryption and storing the encryption master key separately from the data.

[0055] FIGURE 25 illustrates utilization of the cryptographic methods and systems of the present invention with a small working group.

[0056] FIGURE 26 is a block diagram of an illustrative physical token security system employing the secure data

parser in accordance with one embodiment of the present invention.

**[0057]** FIGURE 27 is a block diagram of an illustrative arrangement in which the secure data parser is integrated into a system in accordance with one embodiment of the present invention.

**[0058]** FIGURE 28 is a block diagram of an illustrative data in motion system in accordance with one embodiment of the present invention.

**[0059]** FIGURE 29 is a block diamgram of another illustrative data in motion system in accordance with one embodiment of the present invention.

**[0060]** FIGURE 30-32 are block diagrams of an illustrative system having the secure data parser integrated in accordance with one embodiment of the present invention.

**[0061]** FIGURE 33 is a process flow diagram of an illustrative process for parsing and splitting data in accordance with one embodiment of the present invention.

**[0062]** FIGURE 34 is a process flow diagram of an illustrative process for restoring portions of data into original data in accordance with one embodiment of the present invention.

**[0063]** FIGURE 35 is a process flow diagram of an illustrative process for splitting data at the bit level in accordance with one embodiment of the present invention.

Detailed Description of the Invention

[0064] One aspect of the present invention is to provide a cryptographic system where one or more secure servers, or a trust engine, stores cryptographic keys and user authentication data. Users access the functionality of conventional cryptographic systems through network access to the trust engine, however, the trust engine does not release actual keys and other authentication data and therefore, the keys and data remain secure. This server-centric storage of keys and authentication data provides for user-independent security, portability, availability, and straightforwardness.

[0065] Because users can be confident in, or trust, the cryptographic system to perform user and document authentication and other cryptographic functions, a wide variety of functionality may be incorporated into the system. For example, the trust engine provider can ensure against agreement repudiation by, for example, authenticating the agreement participants, digitally signing the agreement on behalf of or for the participants, and storing a record of the agreement digitally signed by each participant. In addition, the cryptographic system may monitor agreements and determine to apply varying degrees of authentication, based on, for example, price, user, vendor, geographic location, place of use, or the like.

[0066] To facilitate a complete understanding of the invention, the remainder of the detailed description describes the invention with reference to the figures, wherein like elements are referenced with like numerals throughout.

[0067] FIGURE 1 illustrates a block diagram of a cryptographic system 100, according to aspects of an embodiment of the invention. As shown in FIGURE 1, the cryptographic system 100 includes a user system 105, a trust engine 110, a certificate authority 115, and a vendor system 120, communicating through a communication link 125.

[0068] According to one embodiment of the invention, the user system 105 comprises a conventional general-purpose computer having one or more microprocessors, such as, for example, an Intel-based processor. Moreover, the user system 105 includes an appropriate operating system, such as, for example, an operating system capable of including graphics or windows, such as Windows, Unix, Linux, or the like. As shown in FIGURE 1, the user system 105 may include a biometric device 107. The biometric device 107 may advantageously capture a user's biometric and transfer the captured biometric to the trust engine 110. According to one embodiment of the invention, the biometric device may advantageously comprise a device having attributes and features similar to those disclosed in U.S. Patent Application No. 08/926,277, filed on September 5, 1997, entitled *"RELIEF OBJECT IMAGE GENERATOR,"* U.S. Patent Application No. 09/558,634, filed on April 26, 2000, entitled *"IMAGING DEVICE FOR A RELIEF OBJECT AND SYSTEM AND METHOD OF USING THE IMAGE DEVICE,"* U.S. Patent Application No. 09/435,011, filed on November 5, 1999, entitled *"RELIEF OBJECT SENSOR ADAPTOR,"* and U.S. Patent Application No. 09/477,943, filed on January

5, 2000, entitled *"PLANAR OPTICAL IMAGE SENSOR AND SYSTEM FOR GENERATING AN ELECTRONIC IMAGE OF A RELIEF OBJECT FOR FINGERPRINT READING,"* all of which are owned by the instant assignee, and all of which are hereby incorporated by reference herein.

**[0069]** In addition, the user system 105 may connect to the communication link 125 through a conventional service provider, such as, for example, a dial up, digital subscriber line (DSL), cable modem, fiber connection, or the like. According to another embodiment, the user system 105 connects the communication link 125 through network connectivity such as, for example, a local or wide area network. According to one embodiment, the operating system includes a TCP/IP stack that handles all incoming and outgoing message traffic passed over the communication link 125.

**[0070]** Although the user system 105 is disclosed with reference to the foregoing embodiments, the invention is not intended to be limited thereby. Rather, a skilled artisan will recognize from the disclosure herein, a wide number of alternatives embodiments of the user system 105, including almost any computing device capable of sending or receiving information from another computer system. For example, the user system 105 may include, but is not limited to, a computer workstation, an interactive television, an interactive kiosk, a personal mobile computing device, such as a digital assistant, mobile phone, laptop, or the like, a wireless communications device, a smartcard, an embedded computing device, or the like, which can interact with the

communication link 125. In such alternative systems, the operating systems will likely differ and be adapted for the particular device. However, according to one embodiment, the operating systems advantageously continue to provide the appropriate communications protocols needed to establish communication with the communication link 125.

**[0071]** FIGURE 1 illustrates the trust engine 110. According to one embodiment, the trust engine 110 comprises one or more secure servers for accessing and storing sensitive information, which may be any type or form of data, such as, but not limited to text, audio, video, user authentication data and public and private cryptographic keys. According to one embodiment, the authentication data includes data designed to uniquely identify a user of the cryptographic system 100. For example, the authentication data may include a user identification number, one or more biometrics, and a series of questions and answers generated by the trust engine 110 or the user, but answered initially by the user at enrollment. The foregoing questions may include demographic data, such as place of birth, address, anniversary, or the like, personal data, such as mother's maiden name, favorite ice cream, or the like, or other data designed to uniquely identify the user. The trust engine 110 compares a user's authentication data associated with a current transaction, to the authentication data provided at an earlier time, such as, for example, during enrollment. The trust engine 110 may advantageously require the user to produce the

authentication data at the time of each transaction, or, the trust engine 110 may advantageously allow the user to periodically produce authentication data, such as at the beginning of a string of transactions or the logging onto a particular vendor website.

**[0072]** According to the embodiment where the user produces biometric data, the user provides a physical characteristic, such as, but not limited to, facial scan, hand scan, ear scan, iris scan, retinal scan, vascular pattern, DNA, a fingerprint, writing or speech, to the biometric device 107. The biometric device advantageously produces an electronic pattern, or biometric, of the physical characteristic. The electronic pattern is transferred through the user system 105 to the trust engine 110 for either enrollment or authentication purposes.

**[0073]** Once the user produces the appropriate authentication data and the trust engine 110 determines a positive match between that authentication data (current authentication data) and the authentication data provided at the time of enrollment (enrollment authentication data), the trust engine 110 provides the user with complete cryptographic functionality. For example, the properly authenticated user may advantageously employ the trust engine 110 to perform hashing, digitally signing, encrypting and decrypting (often together referred to only as encrypting), creating or distributing digital certificates, and the like. However, the private cryptographic keys used in the cryptographic functions

will not be available outside the trust engine 110, thereby ensuring the integrity of the cryptographic keys.

**[0074]** According to one embodiment, the trust engine 110 generates and stores cryptographic keys. According to another embodiment, at least one cryptographic key is associated with each user. Moreover, when the cryptographic keys include public-key technology, each private key associated with a user is generated within, and not released from, the trust engine 110. Thus, so long as the user has access to the trust engine 110, the user may perform cryptographic functions using his or her private or public key. Such remote access advantageously allows users to remain completely mobile and access cryptographic functionality through practically any Internet connection, such as cellular and satellite phones, kiosks, laptops, hotel rooms and the like.

**[0075]** According to another embodiment, the trust engine 110 performs the cryptographic functionality using a key pair generated for the trust engine 110. According to this embodiment, the trust engine 110 first authenticates the user, and after the user has properly produced authentication data matching the enrollment authentication data, the trust engine 110 uses its own cryptographic key pair to perform cryptographic functions on behalf of the authenticated user.

**[0076]** A skilled artisan will recognize from the disclosure herein that the cryptographic keys may advantageously include some or all of symmetric keys, public keys, and private keys. In addition, a skilled artisan will recognize from the disclosure herein that

the foregoing keys may be implemented with a wide number of algorithms available from commercial technologies, such as, for example, RSA, ELGAMAL, or the like.

**[0077]** FIGURE 1 also illustrates the certificate authority 115. According to one embodiment, the certificate authority 115 may advantageously comprise a trusted third-party organization or company that issues digital certificates, such as, for example, VeriSign, Baltimore, Entrust, or the like. The trust engine 110 may advantageously transmit requests for digital certificates, through one or more conventional digital certificate protocols, such as, for example, PKCS10, to the certificate authority 115. In response, the certificate authority 115 will issue a digital certificate in one or more of a number of differing protocols, such as, for example, PKCS7. According to one embodiment of the invention, the trust engine 110 requests digital certificates from several or all of the prominent certificate authorities 115 such that the trust engine 110 has access to a digital certificate corresponding to the certificate standard of any requesting party.

**[0078]** According to another embodiment, the trust engine 110 internally performs certificate issuances. In this embodiment, the trust engine 110 may access a certificate system for generating certificates and/or may internally generate certificates when they are requested, such as, for example, at the time of key generation or in the certificate standard requested at the time of the

request. The trust engine 110 will be disclosed in greater detail below.

**[0079]** FIGURE 1 also illustrates the vendor system 120. According to one embodiment, the vendor system 120 advantageously comprises a Web server. Typical Web servers generally serve content over the Internet using one of several internet markup languages or document format standards, such as the Hyper-Text Markup Language (HTML) or the Extensible Markup Language (XML). The Web server accepts requests from browsers like Netscape and Internet Explorer and then returns the appropriate electronic documents. A number of server or client-side technologies can be used to increase the power of the Web server beyond its ability to deliver standard electronic documents. For example, these technologies include Common Gateway Interface (CGI) scripts, Secure Sockets Layer (SSL) security, and Active Server Pages (ASPs). The vendor system 120 may advantageously provide electronic content relating to commercial, personal, educational, or other transactions.

**[0080]** Although the vendor system 120 is disclosed with reference to the foregoing embodiments, the invention is not intended to be limited thereby. Rather, a skilled artisan will recognize from the disclosure herein that the vendor system 120 may advantageously comprise any of the devices described with reference to the user system 105 or combination thereof.

**[0081]** FIGURE 1 also illustrates the communication link 125 connecting the user system 105, the trust engine 110, the certificate authority 115, and the vendor system

120. According to one embodiment, the communication link 125 preferably comprises the Internet. The Internet, as used throughout this disclosure is a global network of computers. The structure of the Internet, which is well known to those of ordinary skill in the art, includes a network backbone with networks branching from the backbone. These branches, in turn, have networks branching from them, and so on. Routers move information packets between network levels, and then from network to network, until the packet reaches the neighborhood of its destination. From the destination, the destination network's host directs the information packet to the appropriate terminal, or node. In one advantageous embodiment, the Internet routing hubs comprise domain name system (DNS) servers using Transmission Control Protocol/Internet Protocol (TCP/IP) as is well known in the art. The routing hubs connect to one or more other routing hubs via high-speed communication links.

**[0082]** One popular part of the Internet is the World Wide Web. The World Wide Web contains different computers, which store documents capable of displaying graphical and textual information. The computers that provide information on the World Wide Web are typically called "websites." A website is defined by an Internet address that has an associated electronic page. The electronic page can be identified by a Uniform Resource Locator (URL). Generally, an electronic page is a document that organizes the presentation of text, graphical images, audio, video, and so forth.

[0083] Although the communication link 125 is disclosed in terms of its preferred embodiment, one of ordinary skill in the art will recognize from the disclosure herein that the communication link 125 may include a wide range of interactive communications links. For example, the communication link 125 may include interactive television networks, telephone networks, wireless data transmission systems, two-way cable systems, customized private or public computer networks, interactive kiosk networks, automatic teller machine networks, direct links, satellite or cellular networks, and the like.

[0084] FIGURE 2 illustrates a block diagram of the trust engine 110 of FIGURE 1 according to aspects of an embodiment of the invention. As shown in FIGURE 2, the trust engine 110 includes a transaction engine 205, a depository 210, an authentication engine 215, and a cryptographic engine 220. According to one embodiment of the invention, the trust engine 110 also includes mass storage 225. As further shown in FIGURE 2, the transaction engine 205 communicates with the depository 210, the authentication engine 215, and the cryptographic engine 220, along with the mass storage 225. In addition, the depository 210 communicates with the authentication engine 215, the cryptographic engine 220, and the mass storage 225. Moreover, the authentication engine 215 communicates with the cryptographic engine 220. According to one embodiment of the invention, some or all of the foregoing communications may advantageously comprise the transmission of XML documents to IP

addresses that correspond to the receiving device. As mentioned in the foregoing, XML documents advantageously allow designers to create their own customized document tags, enabling the definition, transmission, validation, and interpretation of data between applications and between organizations. Moreover, some or all of the foregoing communications may include conventional SSL technologies.

[0085] According to one embodiment, the transaction engine 205 comprises a data routing device, such as a conventional Web server available from Netscape, Microsoft, Apache, or the like. For example, the Web server may advantageously receive incoming data from the communication link 125. According to one embodiment of the invention, the incoming data is addressed to a front-end security system for the trust engine 110. For example, the front-end security system may advantageously include a firewall, an intrusion detection system searching for known attack profiles, and/or a virus scanner. After clearing the front-end security system, the data is received by the transaction engine 205 and routed to one of the depository 210, the authentication engine 215, the cryptographic engine 220, and the mass storage 225. In addition, the transaction engine 205 monitors incoming data from the authentication engine 215 and cryptographic engine 220, and routes the data to particular systems through the communication link 125. For example, the transaction engine 205 may advantageously route data to the user system 105, the certificate authority 115, or the vendor system 120.

[0086] According to one embodiment, the data is routed using conventional HTTP routing techniques, such as, for example, employing URLs or Uniform Resource Indicators (URIs). URIs are similar to URLs, however, URIs typically indicate the source of files or actions, such as, for example, executables, scripts, and the like. Therefore, according to the one embodiment, the user system 105, the certificate authority 115, the vendor system 120, and the components of the trust engine 210, advantageously include sufficient data within communication URLs or URIs for the transaction engine 205 to properly route data throughout the cryptographic system.

[0087] Although the data routing is disclosed with reference to its preferred embodiment, a skilled artisan will recognize a wide number of possible data routing solutions or strategies. For example, XML or other data packets may advantageously be unpacked and recognized by their format, content, or the like, such that the transaction engine 205 may properly route data throughout the trust engine 110. Moreover, a skilled artisan will recognize that the data routing may advantageously be adapted to the data transfer protocols conforming to particular network systems, such as, for example, when the communication link 125 comprises a local network.

[0088] According to yet another embodiment of the invention, the transaction engine 205 includes conventional SSL encryption technologies, such that the foregoing systems may authenticate themselves, and vise-versa, with transaction engine 205, during

particular communications. As will be used throughout this disclosure, the term "½ SSL" refers to communications where a server but not necessarily the client, is SSL authenticated, and the term "FULL SSL" refers to communications where the client and the server are SSL authenticated. When the instant disclosure uses the term "SSL", the communication may comprise ½ or FULL SSL.

**[0089]** As the transaction engine 205 routes data to the various components of the cryptographic system 100, the transaction engine 205 may advantageously create an audit trail. According to one embodiment, the audit trail includes a record of at least the type and format of data routed by the transaction engine 205 throughout the cryptographic system 100. Such audit data may advantageously be stored in the mass storage 225.

**[0090]** FIGURE 2 also illustrates the depository 210. According to one embodiment, the depository 210 comprises one or more data storage facilities, such as, for example, a directory server, a database server, or the like. As shown in FIGURE 2, the depository 210 stores cryptographic keys and enrollment authentication data. The cryptographic keys may advantageously correspond to the trust engine 110 or to users of the cryptographic system 100, such as the user or vendor. The enrollment authentication data may advantageously include data designed to uniquely identify a user, such as, user ID, passwords, answers to questions, biometric data, or the like. This enrollment authentication data may advantageously be acquired at enrollment of a user or

another alternative later time. For example, the trust engine 110 may include periodic or other renewal or reissue of enrollment authentication data.

[0091] According to one embodiment, the communication from the transaction engine 205 to and from the authentication engine 215 and the cryptographic engine 220 comprises secure communication, such as, for example conventional SSL technology. In addition, as mentioned in the foregoing, the data of the communications to and from the depository 210 may be transferred using URLs, URIs, HTTP or XML documents, with any of the foregoing advantageously having data requests and formats embedded therein.

[0092] As mentioned above, the depository 210 may advantageously comprises a plurality of secure data storage facilities. In such an embodiment, the secure data storage facilities may be configured such that a compromise of the security in one individual data storage facility will not compromise the cryptographic keys or the authentication data stored therein. For example, according to this embodiment, the cryptographic keys and the authentication data are mathematically operated on so as to statistically and substantially randomize the data stored in each data storage facility. According to one embodiment, the randomization of the data of an individual data storage facility renders that data undecipherable. Thus, compromise of an individual data storage facility produces only a randomized undecipherable number and does not compromise the

security of any cryptographic keys or the authentication data as a whole.

[0093] FIGURE 2 also illustrates the trust engine 110 including the authentication engine 215. According to one embodiment, the authentication engine 215 comprises a data comparator configured to compare data from the transaction engine 205 with data from the depository 210. For example, during authentication, a user supplies current authentication data to the trust engine 110 such that the transaction engine 205 receives the current authentication data. As mentioned in the foregoing, the transaction engine 205 recognizes the data requests, preferably in the URL or URI, and routes the authentication data to the authentication engine 215. Moreover, upon request, the depository 210 forwards enrollment authentication data corresponding to the user to the authentication engine 215. Thus, the authentication engine 215 has both the current authentication data and the enrollment authentication data for comparison.

[0094] According to one embodiment, the communications to the authentication engine comprise secure communications, such as, for example, SSL technology. Additionally, security can be provided within the trust engine 110 components, such as, for example, super-encryption using public key technologies. For example, according to one embodiment, the user encrypts the current authentication data with the public key of the authentication engine 215. In addition, the depository 210 also encrypts the enrollment

authentication data with the public key of the authentication engine 215. In this way, only the authentication engine's private key can be used to decrypt the transmissions.

**[0095]** As shown in FIGURE 2, the trust engine 110 also includes the cryptographic engine 220. According to one embodiment, the cryptographic engine comprises a cryptographic handling module, configured to advantageously provide conventional cryptographic functions, such as, for example, public-key infrastructure (PKI) functionality. For example, the cryptographic engine 220 may advantageously issue public and private keys for users of the cryptographic system 100. In this manner, the cryptographic keys are generated at the cryptographic engine 220 and forwarded to the depository 210 such that at least the private cryptographic keys are not available outside of the trust engine 110. According to another embodiment, the cryptographic engine 220 randomizes and splits at least the private cryptographic key data, thereby storing only the randomized split data. Similar to the splitting of the enrollment authentication data, the splitting process ensures the stored keys are not available outside the cryptographic engine 220. According to another embodiment, the functions of the cryptographic engine can be combined with and performed by the authentication engine 215.

**[0096]** According to one embodiment, communications to and from the cryptographic engine include secure communications, such as SSL technology. In addition, XML

documents may advantageously be employed to transfer data and/or make cryptographic function requests.

[0097] FIGURE 2 also illustrates the trust engine 110 having the mass storage 225. As mentioned in the foregoing, the transaction engine 205 keeps data corresponding to an audit trail and stores such data in the mass storage 225. Similarly, according to one embodiment of the invention, the depository 210 keeps data corresponding to an audit trail and stores such data in the mass storage device 225. The depository audit trail data is similar to that of the transaction engine 205 in that the audit trail data comprises a record of the requests received by the depository 210 and the response thereof. In addition, the mass storage 225 may be used to store digital certificates having the public key of a user contained therein.

[0098] Although the trust engine 110 is disclosed with reference to its preferred and alternative embodiments, the invention is not intended to be limited thereby. Rather, a skilled artisan will recognize in the disclosure herein, a wide number of alternatives for the trust engine 110. For example, the trust engine 110, may advantageously perform only authentication, or alternatively, only some or all of the cryptographic functions, such as data encryption and decryption. According to such embodiments, one of the authentication engine 215 and the cryptographic engine 220 may advantageously be removed, thereby creating a more straightforward design for the trust engine 110. In addition, the cryptographic engine 220 may also

communicate with a certificate authority such that the certificate authority is embodied within the trust engine 110. According to yet another embodiment, the trust engine 110 may advantageously perform authentication and one or more cryptographic functions, such as, for example, digital signing.

**[0099]** FIGURE 3 illustrates a block diagram of the transaction engine 205 of FIGURE 2, according to aspects of an embodiment of the invention. According to this embodiment, the transaction engine 205 comprises an operating system 305 having a handling thread and a listening thread. The operating system 305 may advantageously be similar to those found in conventional high volume servers, such as, for example, Web servers available from Apache. The listening thread monitors the incoming communication from one of the communication link 125, the authentication engine 215, and the cryptographic engine 220 for incoming data flow. The handling thread recognizes particular data structures of the incoming data flow, such as, for example, the foregoing data structures, thereby routing the incoming data to one of the communication link 125, the depository 210, the authentication engine 215, the cryptographic engine 220, or the mass storage 225. As shown in FIGURE 3, the incoming and outgoing data may advantageously be secured through, for example, SSL technology.

**[0100]** FIGURE 4 illustrates a block diagram of the depository 210 of FIGURE 2 according to aspects of an embodiment of the invention. According to this embodiment, the depository 210 comprises one or more

lightweight directory access protocol (LDAP) servers. LDAP directory servers are available from a wide variety of manufacturers such as Netscape, ISO, and others. FIGURE 4 also shows that the directory server preferably stores data 405 corresponding to the cryptographic keys and data 410 corresponding to the enrollment authentication data. According to one embodiment, the depository 210 comprises a single logical memory structure indexing authentication data and cryptographic key data to a unique user ID. The single logical memory structure preferably includes mechanisms to ensure a high degree of trust, or security, in the data stored therein. For example, the physical location of the depository 210 may advantageously include a wide number of conventional security measures, such as limited employee access, modern surveillance systems, and the like. In addition to, or in lieu of, the physical securities, the computer system or server may advantageously include software solutions to protect the stored data. For example, the depository 210 may advantageously create and store data 415 corresponding to an audit trail of actions taken. In addition, the incoming and outgoing communications may advantageously be encrypted with public key encryption coupled with conventional SSL technologies.

**[0101]** According to another embodiment, the depository 210 may comprise distinct and physically separated data storage facilities, as disclosed further with reference to FIGURE 7.

**[0102]** FIGURE 5 illustrates a block diagram of the authentication engine 215 of FIGURE 2 according to

aspects of an embodiment of the invention. Similar to the transaction engine 205 of FIGURE 3, the authentication engine 215 comprises an operating system 505 having at least a listening and a handling thread of a modified version of a conventional Web server, such as, for example, Web servers available from Apache. As shown in FIGURE 5, the authentication engine 215 includes access to at least one private key 510. The private key 510 may advantageously be used for example, to decrypt data from the transaction engine 205 or the depository 210, which was encrypted with a corresponding public key of the authentication engine 215.

**[0103]** FIGURE 5 also illustrates the authentication engine 215 comprising a comparator 515, a data splitting module 520, and a data assembling module 525. According to the preferred embodiment of the invention, the comparator 515 includes technology capable of comparing potentially complex patterns related to the foregoing biometric authentication data. The technology may include hardware, software, or combined solutions for pattern comparisons, such as, for example, those representing finger print patterns or voice patterns. In addition, according to one embodiment, the comparator 515 of the authentication engine 215 may advantageously compare conventional hashes of documents in order to render a comparison result. According to one embodiment of the invention, the comparator 515 includes the application of heuristics 530 to the comparison. The heuristics 530 may advantageously address circumstances surrounding an authentication attempt, such as, for

example, the time of day, IP address or subnet mask, purchasing profile, email address, processor serial number or ID, or the like.

**[0104]** Moreover, the nature of biometric data comparisons may result in varying degrees of confidence being produced from the matching of current biometric authentication data to enrollment data. For example, unlike a traditional password which may only return a positive or negative match, a fingerprint may be determined to be a partial match, e.g. a 90% match, a 75% match, or a 10% match, rather than simply being correct or incorrect. Other biometric identifiers such as voice print analysis or face recognition may share this property of probabilistic authentication, rather than absolute authentication.

**[0105]** When working with such probabilistic authentication or in other cases where an authentication is considered less than absolutely reliable, it is desirable to apply the heuristics 530 to determine whether the level of confidence in the authentication provided is sufficiently high to authenticate the transaction which is being made.

**[0106]** It will sometimes be the case that the transaction at issue is a relatively low value transaction where it is acceptable to be authenticated to a lower level of confidence. This could include a transaction which has a low dollar value associated with it (e.g., a $10 purchase) or a transaction with low risk (e.g., admission to a members-only web site).

[0107] Conversely, for authenticating other transactions, it may be desirable to require a high degree of confidence in the authentication before allowing the transaction to proceed. Such transactions may include transactions of large dollar value (e.g., signing a multi-million dollar supply contract) or transaction with a high risk if an improper authentication occurs (e.g., remotely logging onto a government computer).

[0108] The use of the heuristics 530 in combination with confidence levels and transactions values may be used as will be described below to allow the comparator to provide a dynamic context-sensitive authentication system.

[0109] According to another embodiment of the invention, the comparator 515 may advantageously track authentication attempts for a particular transaction. For example, when a transaction fails, the trust engine 110 may request the user to re-enter his or her current authentication data. The comparator 515 of the authentication engine 215 may advantageously employ an attempt limiter 535 to limit the number of authentication attempts, thereby prohibiting brute-force attempts to impersonate a user's authentication data. According to one embodiment, the attempt limiter 535 comprises a software module monitoring transactions for repeating authentication attempts and, for example, limiting the authentication attempts for a given transaction to three. Thus, the attempt limiter 535 will limit an automated attempt to impersonate an individual's authentication

data to, for example, simply three "guesses." Upon three failures, the attempt limiter 535 may advantageously deny additional authentication attempts. Such denial may advantageously be implemented through, for example, the comparator 515 returning a negative result regardless of the current authentication data being transmitted. On the other hand, the transaction engine 205 may advantageously block any additional authentication attempts pertaining to a transaction in which three attempts have previously failed.

**[0110]** The authentication engine 215 also includes the data splitting module 520 and the data assembling module 525. The data splitting module 520 advantageously comprises a software, hardware, or combination module having the ability to mathematically operate on various data so as to substantially randomize and split the data into portions. According to one embodiment, original data is not recreatable from an individual portion. The data assembling module 525 advantageously comprises a software, hardware, or combination module configured to mathematically operate on the foregoing substantially randomized portions, such that the combination thereof provides the original deciphered data. According to one embodiment, the authentication engine 215 employs the data splitting module 520 to randomize and split enrollment authentication data into portions, and employs the data assembling module 525 to reassemble the portions into usable enrollment authentication data.

**[0111]** FIGURE 6 illustrates a block diagram of the cryptographic engine 220 of the trust engine 200 of

FIGURE 2 according to aspects of one embodiment of the invention. Similar to the transaction engine 205 of FIGURE 3, the cryptographic engine 220 comprises an operating system 605 having at least a listening and a handling thread of a modified version of a conventional Web server, such as, for example, Web servers available from Apache. As shown in FIGURE 6, the cryptographic engine 220 comprises a data splitting module 610 and a data assembling module 620 that function similar to those of FIGURE 5. However, according to one embodiment, the data splitting module 610 and the data assembling module 620 process cryptographic key data, as opposed to the foregoing enrollment authentication data. Although, a skilled artisan will recognize from the disclosure herein that the data splitting module 910 and the data splitting module 620 may be combined with those of the authentication engine 215.

**[0112]** The cryptographic engine 220 also comprises a cryptographic handling module 625 configured to perform one, some or all of a wide number of cryptographic functions. According to one embodiment, the cryptographic handling module 625 may comprise software modules or programs, hardware, or both. According to another embodiment, the cryptographic handling module 625 may perform data comparisons, data parsing, data splitting, data separating, data hashing, data encryption or decryption, digital signature verification or creation, digital certificate generation, storage, or requests, cryptographic key generation, or the like. Moreover, a skilled artisan will recognize from the

disclosure herein that the cryptographic handling module 825 may advantageously comprises a public-key infrastructure, such as Pretty Good Privacy (PGP), an RSA-based public-key system, or a wide number of alternative key management systems. In addition, the cryptographic handling module 625 may perform public-key encryption, symmetric-key encryption, or both. In addition to the foregoing, the cryptographic handling module 625 may include one or more computer programs or modules, hardware, or both, for implementing seamless, transparent, interoperability functions.

**[0113]** A skilled artisan will also recognize from the disclosure herein that the cryptographic functionality may include a wide number or variety of functions generally relating to cryptographic key management systems.

**[0114]** FIGURE 7 illustrates a simplified block diagram of a depository system 700 according to aspects of an embodiment of the invention. As shown in FIGURE 7, the depository system 700 advantageously comprises multiple data storage facilities, for example, data storage facilities D1, D2, D3, and D4. However, it is readily understood by those of ordinary skill in the art that the depository system may have only one data storage facility. According to one embodiment of the invention, each of the data storage facilities D1 through D4 may advantageously comprise some or all of the elements disclosed with reference to the depository 210 of FIGURE 4. Similar to the depository 210, the data storage facilities D1 through D4 communicate with the transaction

engine 205, the authentication engine 215, and the cryptographic engine 220, preferably through conventional SSL. Communication links transferring, for example, XML documents. Communications from the transaction engine 205 may advantageously include requests for data, wherein the request is advantageously broadcast to the IP address of each data storage facility D1 through D4. On the other hand, the transaction engine 205 may broadcast requests to particular data storage facilities based on a wide number of criteria, such as, for example, response time, server loads, maintenance schedules, or the like.

**[0115]** In response to requests for data from the transaction engine 205, the depository system 700 advantageously forwards stored data to the authentication engine 215 and the cryptographic engine 220. The respective data assembling modules receive the forwarded data and assemble the data into useable formats. On the other hand, communications from the authentication engine 215 and the cryptographic engine 220 to the data storage facilities D1 through D4 may include the transmission of sensitive data to be stored. For example, according to one embodiment, the authentication engine 215 and the cryptographic engine 220 may advantageously employ their respective data splitting modules to divide sensitive data into undecipherable portions, and then transmit one or more undecipherable portions of the sensitive data to a particular data storage facility.

**[0116]** According to one embodiment, each data storage facility, D1 through D4, comprises a separate and independent storage system, such as, for example, a

directory server. According to another embodiment of the invention, the depository system 700 comprises multiple geographically separated independent data storage systems. By distributing the sensitive data into distinct and independent storage facilities D1 through D4, some or all of which may be advantageously geographically separated, the depository system 700 provides redundancy along with additional security measures. For example, according to one embodiment, only data from two of the multiple data storage facilities, D1 through D4, are needed to decipher and reassemble the sensitive data. Thus, as many as two of the four data storage facilities D1 through D4 may be inoperative due to maintenance, system failure, power failure, or the like, without affecting the functionality of the trust engine 110. In addition, because, according to one embodiment, the data stored in each data storage facility is randomized and undecipherable, compromise of any individual data storage facility does not necessarily compromise the sensitive data. Moreover, in the embodiment having geographical separation of the data storage facilities, a compromise of multiple geographically remote facilities becomes increasingly difficult. In fact, even a rogue employee will be greatly challenged to subvert the needed multiple independent geographically remote data storage facilities.

**[0117]** Although the depository system 700 is disclosed with reference to its preferred and alternative embodiments, the invention is not intended to be limited

thereby. Rather, a skilled artisan will recognize from the disclosure herein, a wide number of alternatives for the depository system 700. For example, the depository system 700 may comprise one, two or more data storage facilities. In addition, sensitive data may be mathematically operated such that portions from two or more data storage facilities are needed to reassemble and decipher the sensitive data.

**[0118]** As mentioned in the foregoing, the authentication engine 215 and the cryptographic engine 220 each include a data splitting module 520 and 610, respectively, for splitting any type or form of sensitive data, such as, for example, text, audio, video, the authentication data and the cryptographic key data. FIGURE 8 illustrates a flowchart of a data splitting process 800 performed by the data splitting module according to aspects of an embodiment of the invention. As shown in FIGURE 8, the data splitting process 800 begins at step 805 when sensitive data "S" is received by the data splitting module of the authentication engine 215 or the cryptographic engine 220. Preferably, in step 810, the data splitting module then generates a substantially random number, value, or string or set of bits, "A." For example, the random number A may be generated in a wide number of varying conventional techniques available to one of ordinary skill in the art, for producing high quality random numbers suitable for use in cryptographic applications. In addition, according to one embodiment, the random number A comprises a bit length which may be any suitable length,

such as shorter, longer or equal to the bit length of the sensitive data, S.

**[0119]** In addition, in step 820 the data splitting process 800 generates another statistically random number "C." According to the preferred embodiment, the generation of the statistically random numbers A and C may advantageously be done in parallel. The data splitting module then combines the numbers A and C with the sensitive data S such that new numbers "B" and "D" are generated. For example, number B may comprise the binary combination of A XOR S and number D may comprise the binary combination of C XOR S. The XOR function, or the "exclusive-or" function, is well known to those of ordinary skill in the art. The foregoing combinations preferably occur in steps 825 and 830, respectively, and, according to one embodiment, the foregoing combinations also occur in parallel. The data splitting process 800 then proceeds to step 835 where the random numbers A and C and the numbers B and D are paired such that none of the pairings contain sufficient data, by themselves, to reorganize and decipher the original sensitive data S. For example, the numbers may be paired as follows: AC, AD, BC, and BD. According to one embodiment, each of the foregoing pairings is distributed to one of the depositories D1 through D4 of FIGURE 7. According to another embodiment, each of the foregoing pairings is randomly distributed to one of the depositories D1 through D4. For example, during a first data splitting process 800, the pairing AC may be sent to depository D2, through, for example, a random selection of D2's IP

address. Then, during a second data splitting process 800, the pairing AC may be sent to depository D4, through, for example, a random selection of D4's IP address. In addition, the pairings may all be stored on one depository, and may be stored in separate locations on said depository.

**[0120]** Based on the foregoing, the data splitting process 800 advantageously places portions of the sensitive data in each of the four data storage facilities D1 through D4, such that no single data storage facility D1 through D4 includes sufficient encrypted data to recreate the original sensitive data S. As mentioned in the foregoing, such randomization of the data into individually unusable encrypted portions increases security and provides for maintained trust in the data even if one of the data storage facilities, D1 through D4, is compromised.

**[0121]** Although the data splitting process 800 is disclosed with reference to its preferred embodiment, the invention is not intended to be limited thereby. Rather a skilled artisan will recognize from the disclosure herein, a wide number of alternatives for the data splitting process 800. For example, the data splitting process may advantageously split the data into two numbers, for example, random number A and number B and, randomly distribute A and B through two data storage facilities. Moreover, the data splitting process 800 may advantageously split the data among a wide number of data storage facilities through generation of additional random numbers. The data may be split into any desired,

selected, predetermined, or randomly assigned size unit, including but not limited to, a bit, bits, bytes, kilobytes, megabytes or larger, or any combination or sequence of sizes. In addition, varying the sizes of the data units resulting from the splitting process may render the data more difficult to restore to a useable form, thereby increasing security of sensitive data. It is readily apparent to those of ordinary skill in the art that the split data unit sizes may be a wide variety of data unit sizes or patterns of sizes or combinations of sizes. For example, the data unit sizes may be selected or predetermined to be all of the same size, a fixed set of different sizes, a combination of sizes, or randomly generates sizes. Similarly, the data units may be distributed into one or more shares according to a fixed or predetermined data unit size, a pattern or combination of data unit sizes, or a randomly generated data unit size or sizes per share.

**[0122]** As mentioned in the foregoing, in order to recreate the sensitive data S, the data portions need to be derandomized and reorganized. This process may advantageously occur in the data assembling modules, 525 and 620, of the authentication engine 215 and the cryptographic engine 220, respectively. The data assembling module, for example, data assembly module 525, receives data portions from the data storage facilities D1 through D4, and reassembles the data into useable form. For example, according to one embodiment where the data splitting module 520 employed the data splitting process 800 of FIGURE 8, the data assembling module 525

uses data portions from at least two of the data storage facilities D1 through D4 to recreate the sensitive data S. For example, the pairings of AC, AD, BC, and BD, were distributed such that any two provide one of A and B, or, C and D. Noting that S = A XOR B or S = C XOR D indicates that when the data assembling module receives one of A and B, or, C and D, the data assembling module 525 can advantageously reassemble the sensitive data S. Thus, the data assembling module 525 may assemble the sensitive data S, when, for example, it receives data portions from at least the first two of the data storage facilities D1 through D4 to respond to an assemble request by the trust engine 110.

**[0123]** Based on the above data splitting and assembling processes, the sensitive data S exists in usable format only in a limited area of the trust engine 110. For example, when the sensitive data S includes enrollment authentication data, usable, nonrandomized enrollment authentication data is available only in the authentication engine 215. Likewise, when the sensitive data S includes private cryptographic key data, usable, nonrandomized private cryptographic key data is available only in the cryptographic engine 220.

**[0124]** Although the data splitting and assembling processes are disclosed with reference to their preferred embodiments, the invention is not intended to be limited thereby. Rather, a skilled artisan will recognize from the disclosure herein, a wide number of alternatives for splitting and reassembling the sensitive data S. For example, public-key encryption may be used to further

secure the data at the data storage facilities D1 through D4. In addition, it is readily apparent to those of ordinary skill in the art that the data splitting module described herein is also a separate and distinct embodiment of the present invention that may be incorporated into, combined with or otherwise made part of any pre-existing computer systems, software suites, database, or combinations thereof, or other embodiments of the present invention, such as the trust engine, authentication engine, and transaction engine disclosed and described herein.

**[0125]** FIGURE 9A illustrates a data flow of an enrollment process 900 according to aspects of an embodiment of the invention. As shown in FIGURE 9A, the enrollment process 900 begins at step 905 when a user desires to enroll with the trust engine 110 of the cryptographic system 100. According to this embodiment, the user system 105 advantageously includes a client-side applet, such as a Java-based, that queries the user to enter enrollment data, such as demographic data and enrollment authentication data. According to one embodiment, the enrollment authentication data includes user ID, password(s), biometric(s), or the like. According to one embodiment, during the querying process, the client-side applet preferably communicates with the trust engine 110 to ensure that a chosen user ID is unique. When the user ID is nonunique, the trust engine 110 may advantageously suggest a unique user ID. The client-side applet gathers the enrollment data and transmits the enrollment data, for example, through and

XML document, to the trust engine 110, and in particular, to the transaction engine 205. According to one embodiment, the transmission is encoded with the public key of the authentication engine 215.

**[0126]** According to one embodiment, the user performs a single enrollment during step 905 of the enrollment process 900. For example, the user enrolls himself or herself as a particular person, such as Joe User. When Joe User desires to enroll as Joe User, CEO of Mega Corp., then according to this embodiment, Joe User enrolls a second time, receives a second unique user ID and the trust engine 110 does not associate the two identities. According to another embodiment of the invention, the enrollment process 900 provides for multiple user identities for a single user ID. Thus, in the above example, the trust engine 110 will advantageously associate the two identities of Joe User. As will be understood by a skilled artisan from the disclosure herein, a user may have many identities, for example, Joe User the head of household, Joe User the member of the Charitable Foundations, and the like. Even though the user may have multiple identities, according to this embodiment, the trust engine 110 preferably stores only one set of enrollment data. Moreover, users may advantageously add, edit/update, or delete identities as they are needed.

**[0127]** Although the enrollment process 900 is disclosed with reference to its preferred embodiment, the invention is not intended to be limited thereby. Rather, a skilled artisan will recognize from the disclosure

herein, a wide number of alternatives for gathering of enrollment data, and in particular, enrollment authentication data. For example, the applet may be common object model (COM) based applet or the like.

**[0128]** On the other hand, the enrollment process may include graded enrollment. For example, at a lowest level of enrollment, the user may enroll over the communication link 125 without producing documentation as to his or her identity. According to an increased level of enrollment, the user enrolls using a trusted third party, such as a digital notary. For example, and the user may appear in person to the trusted third party, produce credentials such as a birth certificate, driver's license, military ID, or the like, and the trusted third party may advantageously include, for example, their digital signature in enrollment submission. The trusted third party may include an actual notary, a government agency, such as the Post Office or Department of Motor Vehicles, a human resources person in a large company enrolling an employee, or the like. A skilled artisan will understand from the disclosure herein that a wide number of varying levels of enrollment may occur during the enrollment process 900.

**[0129]** After receiving the enrollment authentication data, at step 915, the transaction engine 205, using conventional FULL SSL technology forwards the enrollment authentication data to the authentication engine 215. In step 920, the authentication engine 215 decrypts the enrollment authentication data using the private key of the authentication engine 215. In addition, the

authentication engine 215 employs the data splitting module to mathematically operate on the enrollment authentication data so as to split the data into at least two independently undecipherable, randomized, numbers. As mentioned in the foregoing, at least two numbers may comprise a statistically random number and a binary X0Red number. In step 925, the authentication engine 215 forwards each portion of the randomized numbers to one of the data storage facilities D1 through D4. As mentioned in the foregoing, the authentication engine 215 may also advantageously randomize which portions are transferred to which depositories.

**[0130]** Often during the enrollment process 900, the user will also desire to have a digital certificate issued such that he or she may receive encrypted documents from others outside the cryptographic system 100. As mentioned in the foregoing, the certificate authority 115 generally issues digital certificates according to one or more of several conventional standards. Generally, the digital certificate includes a public key of the user or system, which is known to everyone.

**[0131]** Whether the user requests a digital certificate at enrollment, or at another time, the request is transferred through the trust engine 110 to the authentication engine 215. According to one embodiment, the request includes an XML document having, for example, the proper name of the user. According to step 935, the authentication engine 215 transfers the request to the

cryptographic engine 220 instructing the cryptographic engine 220 to generate a cryptographic key or key pair.

**[0132]** Upon request, at step 935, the cryptographic engine 220 generates at least one cryptographic key. According to one embodiment, the cryptographic handling module 625 generates a key pair, where one key is used as a private key, and one is used as a public key. The cryptographic engine 220 stores the private key and, according to one embodiment, a copy of the public key. In step 945, the cryptographic engine 220 transmits a request for a digital certificate to the transaction engine 205. According to one embodiment, the request advantageously includes a standardized request, such as PKCS10, embedded in, for example, an XML document. The request for a digital certificate may advantageously correspond to one or more certificate authorities and the one or more standard formats the certificate authorities require.

**[0133]** In step 950 the transaction engine 205 forwards this request to the certificate authority 115, who, in step 955, returns a digital certificate. The return digital certificate may advantageously be in a standardized format, such as PKCS7, or in a proprietary format of one or more of the certificate authorities 115. In step 960, the digital certificate is received by the transaction engine 205, and a copy is forwarded to the user and a copy is stored with the trust engine 110. The trust engine 110 stores a copy of the certificate such that the trust engine 110 will not need to rely on the availability of the certificate authority 115. For

example, when the user desires to send a digital certificate, or a third party requests the user's digital certificate, the request for the digital certificate is typically sent to the certificate authority 115. However, if the certificate authority 115 is conducting maintenance or has been victim of a failure or security compromise, the digital certificate may not be available.

**[0134]** At any time after issuing the cryptographic keys, the cryptographic engine 220 may advantageously employ the data splitting process 800 described above such that the cryptographic keys are split into independently undecipherable randomized numbers. Similar to the authentication data, at step 965 the cryptographic engine 220 transfers the randomized numbers to the data storage facilities D1 through D4.

**[0135]** A skilled artisan will recognize from the disclosure herein that the user may request a digital certificate anytime after enrollment. Moreover, the communications between systems may advantageously include FULL SSL or public-key encryption technologies. Moreover, the enrollment process may issue multiple digital certificates from multiple certificate authorities, including one or more proprietary certificate authorities internal or external to the trust engine 110.

**[0136]** As disclosed in steps 935 through 960, one embodiment of the invention includes the request for a certificate that is eventually stored on the trust engine 110. Because, according to one embodiment, the cryptographic handling module 625 issues the keys used by

the trust engine 110, each certificate corresponds to a private key. Therefore, the trust engine 110 may advantageously provide for interoperability through monitoring the certificates owned by, or associated with, a user. For example, when the cryptographic engine 220 receives a request for a cryptographic function, the cryptographic handling module 625 may investigate the certificates owned by the requesting user to determine whether the user owns a private key matching the attributes of the request. When such a certificate exists, the cryptographic handling module 625 may use the certificate or the public or private keys associated therewith, to perform the requested function. When such a certificate does not exist, the cryptographic handling module 625 may advantageously and transparently perform a number of actions to attempt to remedy the lack of an appropriate key. For example, FIGURE 9B illustrates a flowchart of an interoperability process 970, which according to aspects of an embodiment of the invention, discloses the foregoing steps to ensure the cryptographic handling module 625 performs cryptographic functions using appropriate keys.

**[0137]** As shown in FIGURE 9B, the interoperability process 970 begins with step 972 where the cryptographic handling module 925 determines the type of certificate desired. According to one embodiment of the invention, the type of certificate may advantageously be specified in the request for cryptographic functions, or other data provided by the requestor. According to another embodiment, the certificate type may be ascertained by

the data format of the request. For example, the cryptographic handling module 925 may advantageously recognize the request corresponds to a particular type.

**[0138]** According to one embodiment, the certificate type may include one or more algorithm standards, for example, RSA, ELGAMAL, or the like. In addition, the certificate type may include one or more key types, such as symmetric keys, public keys, strong encryption keys such as 256 bit keys, less secure keys, or the like. Moreover, the certificate type may include upgrades or replacements of one or more of the foregoing algorithm standards or keys, one or more message or data formats, one or more data encapsulation or encoding schemes, such as Base 32 or Base 64. The certificate type may also include compatibility with one or more third-party cryptographic applications or interfaces, one or more communication protocols, or one or more certificate standards or protocols. A skilled artisan will recognize from the disclosure herein that other differences may exist in certificate types, and translations to and from those differences may be implemented as disclosed herein.

**[0139]** Once the cryptographic handling module 625 determines the certificate type, the interoperability process 970 proceeds to step 974, and determines whether the user owns a certificate matching the type determined in step 974. When the user owns a matching certificate, for example, the trust engine 110 has access to the matching certificate through, for example, prior storage thereof, the cryptographic handling module 825 knows that a matching private key is also stored within the trust

engine 110. For example, the matching private key may be stored within the depository 210 or depository system 700. The cryptographic handling module 625 may advantageously request the matching private key be assembled from, for example, the depository 210, and then in step 976, use the matching private key to perform cryptographic actions or functions. For example, as mentioned in the foregoing, the cryptographic handling module 625 may advantageously perform hashing, hash comparisons, data encryption or decryption, digital signature verification or creation, or the like.

**[0140]** When the user does not own a matching certificate, the interoperability process 970 proceeds to step 978 where the cryptographic handling module 625 determines whether the users owns a cross-certified certificate. According to one embodiment, cross-certification between certificate authorities occurs when a first certificate authority determines to trust certificates from a second certificate authority. In other words, the first certificate authority determines that certificates from the second certificate authority meets certain quality standards, and therefore, may be "certified" as equivalent to the first certificate authority's own certificates. Cross-certification becomes more complex when the certificate authorities issue, for example, certificates having levels of trust. For example, the first certificate authority may provide three levels of trust for a particular certificate, usually based on the degree of reliability in the enrollment process, while the second certificate

authority may provide seven levels of trust. Cross-certification may advantageously track which levels and which certificates from the second certificate authority may be substituted for which levels and which certificates from the first. When the foregoing cross-certification is done officially and publicly between two certification authorities, the mapping of certificates and levels to one another is often called "chaining."

**[0141]** According to another embodiment of the invention, the cryptographic handling module 625 may advantageously develop cross-certifications outside those agreed upon by the certificate authorities. For example, the cryptographic handling module 625 may access a first certificate authority's certificate practice statement (CPS), or other published policy statement, and using, for example, the authentication tokens required by particular trust levels, match the first certificate authority's certificates to those of another certificate authority.

**[0142]** When, in step 978, the cryptographic handling module 625 determines that the users owns a cross-certified certificate, the interoperability process 970 proceeds to step 976, and performs the cryptographic action or function using the cross-certified public key, private key, or both. Alternatively, when the cryptographic handling module 625 determines that the users does not own a cross-certified certificate, the interoperability process 970 proceeds to step 980, where the cryptographic handling module 625 selects a

certificate authority that issues the requested certificate type, or a certificate cross-certified thereto. In step 982, the cryptographic handling module 625 determines whether the user enrollment authentication data, discussed in the foregoing, meets the authentication requirements of the chosen certificate authority. For example, if the user enrolled over a network by, for example, answering demographic and other questions, the authentication data provided may establish a lower level of trust than a user providing biometric data and appearing before a third-party, such as, for example, a notary. According to one embodiment, the foregoing authentication requirements may advantageously be provided in the chosen authentication authority's CPS.

**[0143]** When the user has provided the trust engine 110 with enrollment authentication data meeting the requirements of chosen certificate authority, the interoperability process 970 proceeds to step 984, where the cryptographic handling module 825 acquires the certificate from the chosen certificate authority. According to one embodiment, the cryptographic handling module 625 acquires the certificate by following steps 945 through 960 of the enrollment process 900. For example, the cryptographic handling module 625 may advantageously employ one or more public keys from one or more of the key pairs already available to the cryptographic engine 220, to request the certificate from the certificate authority. According to another embodiment, the cryptographic handling module 625 may advantageously generate one or more new key pairs, and

use the public keys corresponding thereto, to request the certificate from the certificate authority.

**[0144]** According to another embodiment, the trust engine 110 may advantageously include one or more certificate issuing modules capable of issuing one or more certificate types. According to this embodiment, the certificate issuing module may provide the foregoing certificate. When the cryptographic handling module 625 acquires the certificate, the interoperability process 970 proceeds to step 976, and performs the cryptographic action or function using the public key, private key, or both corresponding to the acquired certificate.

**[0145]** When the user, in step 982, has not provided the trust engine 110 with enrollment authentication data meeting the requirements of chosen certificate authority, the cryptographic handling module 625 determines, in step 986 whether there are other certificate authorities that have different authentication requirements. For example, the cryptographic handling module 625 may look for certificate authorities having lower authentication requirements, but still issue the chosen certificates, or cross-certifications thereof.

**[0146]** When the foregoing certificate authority having lower requirements exists, the interoperability process 970 proceeds to step 980 and chooses that certificate authority. Alternatively, when no such certificate authority exists, in step 988, the trust engine 110 may request additional authentication tokens from the user. For example, the trust engine 110 may request new enrollment authentication data comprising, for example,

biometric data. Also, the trust engine 110 may request the user appear before a trusted third party and provide appropriate authenticating credentials, such as, for example, appearing before a notary with a drivers license, social security card, bank card, birth certificate, military ID, or the like. When the trust engine 110 receives updated authentication data, the interoperability process 970 proceeds to step 984 and acquires the foregoing chosen certificate.

**[0147]** Through the foregoing interoperability process 970, the cryptographic handling module 625 advantageously provides seamless, transparent, translations and conversions between differing cryptographic systems. A skilled artisan will recognize from the disclosure herein, a wide number of advantages and implementations of the foregoing interoperable system. For example, the foregoing step 986 of the interoperability process 970 may advantageously include aspects of trust arbitrage, discussed in further detail below, where the certificate authority may under special circumstances accept lower levels of cross-certification. In addition, the interoperability process 970 may include ensuring interoperability between and employment of standard certificate revocations, such as employing certificate revocation lists (CRL), online certificate status protocols (OCSP), or the like.

**[0148]** FIGURE 10 illustrates a data flow of an authentication process 1000 according to aspects of an embodiment of the invention. According to one embodiment, the authentication process 1000 includes

gathering current authentication data from a user and comparing that to the enrollment authentication data of the user. For example, the authentication process 1000 begins at step 1005 where a user desires to perform a transaction with, for example, a vendor. Such transactions may include, for example, selecting a purchase option, requesting access to a restricted area or device of the vendor system 120, or the like. At step 1010, a vendor provides the user with a transaction ID and an authentication request. The transaction ID may advantageously include a 192 bit quantity having a 32 bit timestamp concatenated with a 128 bit random quantity, or a "nonce," concatenated with a 32 bit vendor specific constant. Such a transaction ID uniquely identifies the transaction such that copycat transactions can be refused by the trust engine 110.

**[0149]** The authentication request may advantageously include what level of authentication is needed for a particular transaction. For example, the vendor may specify a particular level of confidence that is required for the transaction at issue. If authentication cannot be made to this level of confidence, as will be discussed below, the transaction will not occur without either further authentication by the user to raise the level of confidence, or a change in the terms of the authentication between the vendor and the server. These issues are discussed more completely below.

**[0150]** According to one embodiment, the transaction ID and the authentication request may be advantageously generated by a vendor-side applet or other software

program. In addition, the transmission of the transaction ID and authentication data may include one or more XML documents encrypted using conventional SSL technology, such as, for example, ½ SSL, or, in other words vendor-side authenticated SSL.

**[0151]** After the user system 105 receives the transaction ID and authentication request, the user system 105 gathers the current authentication data, potentially including current biometric information, from the user. The user system 105, at step 1015, encrypts at least the current authentication data "B" and the transaction ID, with the public key of the authentication engine 215, and transfers that data to the trust engine 110. The transmission preferably comprises XML documents encrypted with at least conventional ½ SSL technology. In step 1020, the transaction engine 205 receives the transmission, preferably recognizes the data format or request in the URL or URI, and forwards the transmission to the authentication engine 215.

**[0152]** During steps 1015 and 1020, the vendor system 120, at step 1025, forwards the transaction ID and the authentication request to the trust engine 110, using the preferred FULL SSL technology. This communication may also include a vendor ID, although vendor identification may also be communicated through a non-random portion of the transaction ID. At steps 1030 and 1035, the transaction engine 205 receives the communication, creates a record in the audit trail, and generates a request for the user's enrollment authentication data to be assembled from the data storage facilities D1 through

D4. At step 1040, the depository system 700 transfers the portions of the enrollment authentication data corresponding to the user to the authentication engine 215. At step 1045, the authentication engine 215 decrypts the transmission using its private key and compares the enrollment authentication data to the current authentication data provided by the user.

**[0153]** The comparison of step 1045 may advantageously apply heuristical context sensitive authentication, as referred to in the forgoing, and discussed in further detail below. For example, if the biometric information received does not match perfectly, a lower confidence match results. In particular embodiments, the level of confidence of the authentication is balanced against the nature of the transaction and the desires of both the user and the vendor. Again, this is discussed in greater detail below.

**[0154]** At step 1050, the authentication engine 215 fills in the authentication request with the result of the comparison of step 1045. According to one embodiment of the invention, the authentication request is filled with a YES/NO or TRUE/FALSE result of the authentication process 1000. In step 1055 the filled-in authentication request is returned to the vendor for the vendor to act upon, for example, allowing the user to complete the transaction that initiated the authentication request. According to one embodiment, a confirmation message is passed to the user.

**[0155]** Based on the foregoing, the authentication process 1000 advantageously keeps sensitive data secure

and produces results configured to maintain the integrity of the sensitive data. For example, the sensitive data is assembled only inside the authentication engine 215. For example, the enrollment authentication data is undecipherable until it is assembled in the authentication engine 215 by the data assembling module, and the current authentication data is undecipherable until it is unwrapped by the conventional SSL technology and the private key of the authentication engine 215. Moreover, the authentication result transmitted to the vendor does not include the sensitive data, and the user may not even know whether he or she produced valid authentication data.

**[0156]** Although the authentication process 1000 is disclosed with reference to its preferred and alternative embodiments, the invention is not intended to be limited thereby. Rather, a skilled artisan will recognize from the disclosure herein, a wide number of alternatives for the authentication process 1000. For example, the vendor may advantageously be replaced by almost any requesting application, even those residing with the user system 105. For example, a client application, such as Microsoft Word, may use an application program interface (API) or a cryptographic API (CAPI) to request authentication before unlocking a document. Alternatively, a mail server, a network, a cellular phone, a personal or mobile computing device, a workstation, or the like, may all make authentication requests that can be filled by the authentication process 1000. In fact, after providing the foregoing trusted

authentication process 1000, the requesting application or device may provide access to or use of a wide number of electronic or computer devices or systems.

**[0157]** Moreover, the authentication process 1000 may employ a wide number of alternative procedures in the event of authentication failure. For example, authentication failure may maintain the same transaction ID and request that the user reenter his or her current authentication data. As mentioned in the foregoing, use of the same transaction ID allows the comparator of the authentication engine 215 to monitor and limit the number of authentication attempts for a particular transaction, thereby creating a more secure cryptographic system 100.

**[0158]** In addition, the authentication process 1000 may be advantageously be employed to develop elegant single sign-on solutions, such as, unlocking a sensitive data vault. For example, successful or positive authentication may provide the authenticated user the ability to automatically access any number of passwords for an almost limitless number of systems and applications. For example, authentication of a user may provide the user access to password, login, financial credentials, or the like, associated with multiple online vendors, a local area network, various personal computing devices, Internet service providers, auction providers, investment brokerages, or the like. By employing a sensitive data vault, users may choose truly large and random passwords because they no longer need to remember them through association. Rather, the authentication process 1000 provides access thereto. For example, a

user may choose a random alphanumeric string that is twenty plus digits in length rather than something associated with a memorable data, name, etc.

**[0159]** According to one embodiment, a sensitive data vault associated with a given user may advantageously be stored in the data storage facilities of the depository 210, or split and stored in the depository system 700. According to this embodiment, after positive user authentication, the trust engine 110 serves the requested sensitive data, such as, for example, to the appropriate password to the requesting application. According to another embodiment, the trust engine 110 may include a separate system for storing the sensitive data vault. For example, the trust engine 110 may include a stand-alone software engine implementing the data vault functionality and figuratively residing "behind" the foregoing front-end security system of the trust engine 110. According to this embodiment, the software engine serves the requested sensitive data after the software engine receives a signal indicating positive user authentication from the trust engine 110.

**[0160]** In yet another embodiment, the data vault may be implemented by a third-party system. Similar to the software engine embodiment, the third-party system may advantageously serve the requested sensitive data after the third-party system receives a signal indicating positive user authentication from the trust engine 110. According to yet another embodiment, the data vault may be implemented on the user system 105. A user-side software engine may advantageously serve the foregoing

data after receiving a signal indicating positive user authentication from the trust engine 110.

**[0161]** Although the foregoing data vaults are disclosed with reference to alternative embodiments, a skilled artisan will recognize from the disclosure herein, a wide number of additional implementations thereof. For example, a particular data vault may include aspects from some or all of the foregoing embodiments. In addition, any of the foregoing data vaults may employ one or more authentication requests at varying times. For example, any of the data vaults may require authentication every one or more transactions, periodically, every one or more sessions, every access to one or more Webpages or Websites, at one or more other specified intervals, or the like.

**[0162]** FIGURE 11 illustrates a data flow of a signing process 1100 according to aspects of an embodiment of the invention. As shown in FIGURE 11, the signing process 1100 includes steps similar to those of the authentication process 1000 described in the foregoing with reference to FIGURE 10. According to one embodiment of the invention, the signing process 1100 first authenticates the user and then performs one or more of several digital signing functions as will be discussed in further detail below. According to another embodiment, the signing process 1100 may advantageously store data related thereto, such as hashes of messages or documents, or the like. This data may advantageously be used in an audit or any other event, such as for example, when a participating party attempts to repudiate a transaction.

**[0163]** As shown in FIGURE 11, during the authentication steps, the user and vendor may advantageously agree on a message, such as, for example, a contract. During signing, the signing process 1100 advantageously ensures that the contract signed by the user is identical to the contract supplied by the vendor. Therefore, according to one embodiment, during authentication, the vendor and the user include a hash of their respective copies of the message or contract, in the data transmitted to the authentication engine 215. By employing only a hash of a message or contract, the trust engine 110 may advantageously store a significantly reduced amount of data, providing for a more efficient and cost effective cryptographic system. In addition, the stored hash may be advantageously compared to a hash of a document in question to determine whether the document in question matches one signed by any of the parties. The ability to determine whether the document is identical to one relating to a transaction provides for additional evidence that can be used against a claim for repudiation by a party to a transaction.

**[0164]** In step 1103, the authentication engine 215 assembles the enrollment authentication data and compares it to the current authentication data provided by the user. When the comparator of the authentication engine 215 indicates that the enrollment authentication data matches the current authentication data, the comparator of the authentication engine 215 also compares the hash of the message supplied by the vendor to the hash of the message supplied by the user. Thus, the authentication

engine 215 advantageously ensures that the message agreed to by the user is identical to that agreed to by the vendor.

**[0165]** In step 1105, the authentication engine 215 transmits a digital signature request to the cryptographic engine 220. According to one embodiment of the invention, the request includes a hash of the message or contract. However, a skilled artisan will recognize from the disclosure herein that the cryptographic engine 220 may encrypt virtually any type of data, including, but not limited to, video, audio, biometrics, images or text to form the desired digital signature. Returning to step 1105, the digital signature request preferably comprises an XML document communicated through conventional SSL technologies.

**[0166]** In step 1110, the authentication engine 215 transmits a request to each of the data storage facilities D1 through D4, such that each of the data storage facilities D1 through D4 transmit their respective portion of the cryptographic key or keys corresponding to a signing party. According to another embodiment, the cryptographic engine 220 employs some or all of the steps of the interoperability process 970 discussed in the foregoing, such that the cryptographic engine 220 first determines the appropriate key or keys to request from the depository 210 or the depository system 700 for the signing party, and takes actions to provide appropriate matching keys. According to still another embodiment, the authentication engine 215 or the cryptographic engine 220 may advantageously request one

or more of the keys associated with the signing party and stored in the depository 210 or depository system 700.

**[0167]** According to one embodiment, the signing party includes one or both the user and the vendor. In such case, the authentication engine 215 advantageously requests the cryptographic keys corresponding to the user and/or the vendor. According to another embodiment, the signing party includes the trust engine 110. In this embodiment, the trust engine 110 is certifying that the authentication process 1000 properly authenticated the user, vendor, or both. Therefore, the authentication engine 215 requests the cryptographic key of the trust engine 110, such as, for example, the key belonging to the cryptographic engine 220, to perform the digital signature. According to another embodiment, the trust engine 110 performs a digital notary-like function. In this embodiment, the signing party includes the user, vendor, or both, along with the trust engine 110. Thus, the trust engine 110 provides the digital signature of the user and/or vendor, and then indicates with its own digital signature that the user and/or vendor were properly authenticated. In this embodiment, the authentication engine 215 may advantageously request assembly of the cryptographic keys corresponding to the user, the vendor, or both. According to another embodiment, the authentication engine 215 may advantageously request assembly of the cryptographic keys corresponding to the trust engine 110.

**[0168]** According to another embodiment, the trust engine 110 performs power of attorney-like functions.

For example, the trust engine 110 may digitally sign the message on behalf of a third party. In such case, the authentication engine 215 requests the cryptographic keys associated with the third party. According to this embodiment, the signing process 1100 may advantageously include authentication of the third party, before allowing power of attorney-like functions. In addition, the authentication process 1000 may include a check for third party constraints, such as, for example, business logic or the like dictating when and in what circumstances a particular third-party's signature may be used.

**[0169]** Based on the foregoing, in step 1110, the authentication engine requested the cryptographic keys from the data storage facilities D1 through D4 corresponding to the signing party. In step 1115, the data storage facilities D1 through D4 transmit their respective portions of the cryptographic key corresponding to the signing party to the cryptographic engine 220. According to one embodiment, the foregoing transmissions include SSL technologies. According to another embodiment, the foregoing transmissions may advantageously be super-encrypted with the public key of the cryptographic engine 220.

**[0170]** In step 1120, the cryptographic engine 220 assembles the foregoing cryptographic keys of the signing party and encrypts the message therewith, thereby forming the digital signature(s). In step 1125 of the signing process 1100, the cryptographic engine 220 transmits the digital signature(s) to the authentication engine 215.

In step 1130, the authentication engine 215 transmits the filled-in authentication request along with a copy of the hashed message and the digital signature(s) to the transaction engine 205. In step 1135, the transaction engine 205 transmits a receipt comprising the transaction ID, an indication of whether the authentication was successful, and the digital signature(s), to the vendor. According to one embodiment, the foregoing transmission may advantageously include the digital signature of the trust engine 110. For example, the trust engine 110 may encrypt the hash of the receipt with its private key, thereby forming a digital signature to be attached to the transmission to the vendor.

**[0171]** According to one embodiment, the transaction engine 205 also transmits a confirmation message to the user. Although the signing process 1100 is disclosed with reference to its preferred and alternative embodiments, the invention is not intended to be limited thereby. Rather, a skilled artisan will recognize from the disclosure herein, a wide number of alternatives for the signing process 1100. For example, the vendor may be replaced with a user application, such as an email application. For example, the user may wish to digitally sign a particular email with his or her digital signature. In such an embodiment, the transmission throughout the signing process 1100 may advantageously include only one copy of a hash of the message. Moreover, a skilled artisan will recognize from the disclosure herein that a wide number of client applications may request digital signatures. For

example, the client applications may comprise word processors, spreadsheets, emails, voicemail, access to restricted system areas, or the like.

**[0172]** In addition, a skilled artisan will recognize from the disclosure herein that steps 1105 through 1120 of the signing process 1100 may advantageously employ some or all of the steps of the interoperability process 970 of FIGURE 9B, thereby providing interoperability between differing cryptographic systems that may, for example, need to process the digital signature under differing signature types.

**[0173]** FIGURE 12 illustrates a data flow of an encryption/decryption process 1200 according to aspects of an embodiment of the invention. As shown in FIGURE 12, the decryption process 1200 begins by authenticating the user using the authentication process 1000. According to one embodiment, the authentication process 1000 includes in the authentication request, a synchronous session key. For example, in conventional PKI technologies, it is understood by skilled artisans that encrypting or decrypting data using public and private keys is mathematically intensive and may require significant system resources. However, in symmetric key cryptographic systems, or systems where the sender and receiver of a message share a single common key that is used to encrypt and decrypt a message, the mathematical operations are significantly simpler and faster. Thus, in the conventional PKI technologies, the sender of a message will generate synchronous session key, and encrypt the message using the simpler, faster symmetric

key system. Then, the sender will encrypt the session key with the public key of the receiver. The encrypted session key will be attached to the synchronously encrypted message and both data are sent to the receiver. The receiver uses his or her private key to decrypt the session key, and then uses the session key to decrypt the message. Based on the foregoing, the simpler and faster symmetric key system is used for the majority of the encryption/decryption processing. Thus, in the decryption process 1200, the decryption advantageously assumes that a synchronous key has been encrypted with the public key of the user. Thus, as mentioned in the foregoing, the encrypted session key is included in the authentication request.

**[0174]** Returning to the decryption process 1200, after the user has been authenticated in step 1205, the authentication engine 215 forwards the encrypted session key to the cryptographic engine 220. In step 1210, the authentication engine 215 forwards a request to each of the data storage facilities, D1 through D4, requesting the cryptographic key data of the user. In step 1215, each data storage facility, D1 through D4, transmits their respective portion of the cryptographic key to the cryptographic engine 220. According to one embodiment, the foregoing transmission is encrypted with the public key of the cryptographic engine 220.

**[0175]** In step 1220 of the decryption process 1200, the cryptographic engine 220 assembles the cryptographic key and decrypts the session key therewith. In step 1225, the cryptographic engine forwards the session key

to the authentication engine 215. In step 1227, the authentication engine 215 fills in the authentication request including the decrypted session key, and transmits the filled-in authentication request to the transaction engine 205. In step 1230, the transaction engine 205 forwards the authentication request along with the session key to the requesting application or vendor. Then, according to one embodiment, the requesting application or vendor uses the session key to decrypt the encrypted message.

**[0176]** Although the decryption process 1200 is disclosed with reference to its preferred and alternative embodiments, a skilled artisan will recognize from the disclosure herein, a wide number of alternatives for the decryption process 1200. For example, the decryption process 1200 may forego synchronous key encryption and rely on full public-key technology. In such an embodiment, the requesting application may transmit the entire message to the cryptographic engine 220, or, may employ some type of compression or reversible hash in order to transmit the message to the cryptographic engine 220. A skilled artisan will also recognize from the disclosure herein that the foregoing communications may advantageously include XML documents wrapped in SSL technology.

**[0177]** The encryption/decryption process 1200 also provides for encryption of documents or other data. Thus, in step 1235, a requesting application or vendor may advantageously transmit to the transaction engine 205 of the trust engine 110, a request for the public key of

the user. The requesting application or vendor makes this request because the requesting application or vendor uses the public key of the user, for example, to encrypt the session key that will be used to encrypt the document or message. As mentioned in the enrollment process 900, the transaction engine 205 stores a copy of the digital certificate of the user, for example, in the mass storage 225. Thus, in step 1240 of the encryption process 1200, the transaction engine 205 requests the digital certificate of the user from the mass storage 225. In step 1245, the mass storage 225 transmits the digital certificate corresponding to the user, to the transaction engine 205. In step 1250, the transaction engine 205 transmits the digital certificate to the requesting application or vendor. According to one embodiment, the encryption portion of the encryption process 1200 does not include the authentication of a user. This is because the requesting vendor needs only the public key of the user, and is not requesting any sensitive data.

**[0178]** A skilled artisan will recognize from the disclosure herein that if a particular user does not have a digital certificate, the trust engine 110 may employ some or all of the enrollment process 900 in order to generate a digital certificate for that particular user. Then, the trust engine 110 may initiate the encryption/decryption process 1200 and thereby provide the appropriate digital certificate. In addition, a skilled artisan will recognize from the disclosure herein that steps 1220 and 1235 through 1250 of the encryption/decryption process 1200 may advantageously

employ some or all of the steps of the interoperability process of FIGURE 9B, thereby providing interoperability between differing cryptographic systems that may, for example, need to process the encryption.

**[0179]** FIGURE 13 illustrates a simplified block diagram of a trust engine system 1300 according to aspects of yet another embodiment of the invention. As shown in FIGURE 13, the trust engine system 1300 comprises a plurality of distinct trust engines 1305, 1310, 1315, and 1320, respectively. To facilitate a more complete understanding of the invention, FIGURE 13 illustrates each trust engine, 1305, 1310, 1315, and 1320 as having a transaction engine, a depository, and an authentication engine. However, a skilled artisan will recognize that each transaction engine may advantageously comprise some, a combination, or all of the elements and communication channels disclosed with reference to FIGURES 1-8. For example, one embodiment may advantageously include trust engines having one or more transaction engines, depositories, and cryptographic servers or any combinations thereof.

**[0180]** According to one embodiment of the invention, each of the trust engines 1305, 1310, 1315 and 1320 are geographically separated, such that, for example, the trust engine 1305 may reside in a first location, the trust engine 1310 may reside in a second location, the trust engine 1315 may reside in a third location, and the trust engine 1320 may reside in a fourth location. The foregoing geographic separation advantageously decreases

system response time while increasing the security of the overall trust engine system 1300.

**[0181]** For example, when a user logs onto the cryptographic system 100, the user may be nearest the first location and may desire to be authenticated. As described with reference to FIGURE 10, to be authenticated, the user provides current authentication data, such as a biometric or the like, and the current authentication data is compared to that user's enrollment authentication data. Therefore, according to one example, the user advantageously provides current authentication data to the geographically nearest trust engine 1305. The transaction engine 1321 of the trust engine 1305 then forwards the current authentication data to the authentication engine 1322 also residing at the first location. According to another embodiment, the transaction engine 1321 forwards the current authentication data to one or more of the authentication engines of the trust engines 1310, 1315, or 1320.

**[0182]** The transaction engine 1321 also requests the assembly of the enrollment authentication data from the depositories of, for example, each of the trust engines, 1305 through 1320. According to this embodiment, each depository provides its portion of the enrollment authentication data to the authentication engine 1322 of the trust engine 1305. The authentication engine 1322 then employs the encrypted data portions from, for example, the first two depositories to respond, and assembles the enrollment authentication data into deciphered form. The authentication engine 1322 compares

the enrollment authentication data with the current authentication data and returns an authentication result to the transaction engine 1321 of the trust engine 1305.

**[0183]** Based on the above, the trust engine system 1300 employs the nearest one of a plurality of geographically separated trust engines, 1305 through 1320, to perform the authentication process. According to one embodiment of the invention, the routing of information to the nearest transaction engine may advantageously be performed at client-side applets executing on one or more of the user system 105, vendor system 120, or certificate authority 115. According to an alternative embodiment, a more sophisticated decision process may be employed to select from the trust engines 1305 through 1320. For example, the decision may be based on the availability, operability, speed of connections, load, performance, geographic proximity, or a combination thereof, of a given trust engine.

**[0184]** In this way, the trust engine system 1300 lowers its response time while maintaining the security advantages associated with geographically remote data storage facilities, such as those discussed with reference to FIGURE 7 where each data storage facility stores randomized portions of sensitive data. For example, a security compromise at, for example, the depository 1325 of the trust engine 1315 does not necessarily compromise the sensitive data of the trust engine system 1300. This is because the depository 1325 contains only non-decipherable randomized data that, without more, is entirely useless.

[0185] According to another embodiment, the trust engine system 1300 may advantageously include multiple cryptographic engines arranged similar to the authentication engines. The cryptographic engines may advantageously perform cryptographic functions such as those disclosed with reference to FIGURES 1-8. According to yet another embodiment, the trust engine system 1300 may advantageously replace the multiple authentication engines with multiple cryptographic engines, thereby performing cryptographic functions such as those disclosed with reference to FIGURES 1-8. According to yet another embodiment of the invention, the trust engine system 1300 may replace each multiple authentication engine with an engine having some or all of the functionality of the authentication engines, cryptographic engines, or both, as disclosed in the foregoing,

[0186] Although the trust engine system 1300 is disclosed with reference to its preferred and alternative embodiments, a skilled artisan will recognize that the trust engine system 1300 may comprise portions of trust engines 1305 through 1320. For example, the trust engine system 1300 may include one or more transaction engines, one or more depositories, one or more authentication engines, or one or more cryptographic engines or combinations thereof.

[0187] FIGURE 14 illustrates a simplified block diagram of a trust engine System 1400 according to aspects of yet another embodiment of the invention. As shown in FIGURE 14, the trust engine system 1400 includes

multiple trust engines 1405, 1410, 1415 and 1420. According to one embodiment, each of the trust engines 1405, 1410, 1415 and 1420, comprise some or all of the elements of trust engine 110 disclosed with reference to FIGURES 1-8. According to this embodiment, when the client side applets of the user system 105, the vendor system 120, or the certificate authority 115, communicate with the trust engine system 1400, those communications are sent to the IP address of each of the trust engines 1405 through 1420. Further, each transaction engine of each of the trust engines, 1405, 1410, 1415, and 1420, behaves similar to the transaction engine 1321 of the trust engine 1305 disclosed with reference to FIGURE 13. For example, during an authentication process, each transaction engine of each of the trust engines 1405, 1410, 1415, and 1420 transmits the current authentication data to their respective authentication engines and transmits a request to assemble the randomized data stored in each of the depositories of each of the trust engines 1405 through 1420. FIGURE 14 does not illustrate all of these communications; as such illustration would become overly complex. Continuing with the authentication process, each of the depositories then communicates its portion of the randomized data to each of the authentication engines of the each of the trust engines 1405 through 1420. Each of the authentication engines of the each of the trust engines employs its comparator to determine whether the current authentication data matches the enrollment authentication data provided by the depositories of each of the trust

engines 1405 through 1420. According to this embodiment, the result of the comparison by each of the authentication engines is then transmitted to a redundancy module of the other three trust engines. For example, the result of the authentication engine from the trust engine 1405 is transmitted to the redundancy modules of the trust engines 1410, 1415, and 1420. Thus, the redundancy module of the trust engine 1405 likewise receives the result of the authentication engines from the trust engines 1410, 1415, and 1420.

**[0188]** FIGURE 15 illustrates a block diagram of the redundancy module of FIGURE 14. The redundancy module comprises a comparator configured to receive the authentication result from three authentication engines and transmit that result to the transaction engine of the fourth trust engine. The comparator compares the authentication result form the three authentication engines, and if two of the results agree, the comparator concludes that the authentication result should match that of the two agreeing authentication engines. This result is then transmitted back to the transaction engine corresponding to the trust engine not associated with the three authentication engines.

**[0189]** Based on the foregoing, the redundancy module determines an authentication result from data received from authentication engines that are preferably geographically remote from the trust engine of that the redundancy module. By providing such redundancy functionality, the trust engine system 1400 ensures that a compromise of the authentication engine of one of the

trust engines 1405 through 1420, is insufficient to compromise the authentication result of the redundancy module of that particular trust engine. A skilled artisan will recognize that redundancy module functionality of the trust engine system 1400 may also be applied to the cryptographic engine of each of the trust engines 1405 through 1420. However, such cryptographic engine communication was not shown in FIGURE 14 to avoid complexity. Moreover, a skilled artisan will recognize a wide number of alternative authentication result conflict resolution algorithms for the comparator of FIGURE 15 are suitable for use in the present invention.

**[0190]** According to yet another embodiment of the invention, the trust engine system 1400 may advantageously employ the redundancy module during cryptographic comparison steps. For example, some or all of the foregoing redundancy module disclosure with reference to FIGURES 14 and 15 may advantageously be implemented during a hash comparison of documents provided by one or more parties during a particular transaction.

**[0191]** Although the foregoing invention has been described in terms of certain preferred and alternative embodiments, other embodiments will be apparent to those of ordinary skill in the art from the disclosure herein. For example, the trust engine 110 may issue short-term certificates, where the private cryptographic key is released to the user for a predetermined period of time. For example, current certificate standards include a validity field that can be set to expire after a

predetermined amount of time. Thus, the trust engine 110 may release a private key to a user where the private key would be valid for, for example, 24 hours. According to such an embodiment, the trust engine 110 may advantageously issue a new cryptographic key pair to be associated with a particular user and then release the private key of the new cryptographic key pair. Then, once the private cryptographic key is released, the trust engine 110 immediately expires any internal valid use of such private key, as it is no longer securable by the trust engine 110.

**[0192]** In addition, a skilled artisan will recognize that the cryptographic system 100 or the trust engine 110 may include the ability to recognize any type of devices, such as, but not limited to, a laptop, a cell phone, a network, a biometric device or the like. According to one embodiment, such recognition may come from data supplied in the request for a particular service, such as, a request for authentication leading to access or use, a request for cryptographic functionality, or the like. According to one embodiment, the foregoing request may include a unique device identifier, such as, for example, a processor ID. Alternatively, the request may include data in a particular recognizable data format. For example, mobile and satellite phones often do not include the processing power for full X509.v3 heavy encryption certificates, and therefore do not request them. According to this embodiment, the trust engine 110 may recognize the type of data format presented, and respond only in kind.

**[0193]** In an additional aspect of the system described above, context sensitive authentication can be provided using various techniques as will be described below. Context sensitive authentication, for example as shown in FIGURE 16, provides the possibility of evaluating not only the actual data which is sent by the user when attempting to authenticate himself, but also the circumstances surrounding the generation and delivery of that data. Such techniques may also support transaction specific trust arbitrage between the user and trust engine 110 or between the vendor and trust engine 110, as will be described below.

**[0194]** As discussed above, authentication is the process of proving that a user is who he says he is. Generally, authentication requires demonstrating some fact to an authentication authority. The trust engine 110 of the present invention represents the authority to which a user must authenticate himself. The user must demonstrate to the trust engine 110 that he is who he says he is by either: knowing something that only the user should know (knowledge-based authentication), having something that only the user should have (token-based authentication), or by being something that only the user should be (biometric-based authentication).

**[0195]** Examples of knowledge-based authentication include without limitation a password, PIN number, or lock combination. Examples of token-based authentication include without limitation a house key, a physical credit card, a driver's license, or a particular phone number. Examples of biometric-based authentication include

without limitation a fingerprint, handwriting analysis, facial scan, hand scan, ear scan, iris scan, vascular pattern, DNA, a voice analysis, or a retinal scan.

**[0196]** Each type of authentication has particular advantages and disadvantages, and each provides a different level of security. For example, it is generally harder to create a false fingerprint that matches someone else's than it is to overhear someone's password and repeat it. Each type of authentication also requires a different type of data to be known to the authenticating authority in order to verify someone using that form of authentication.

**[0197]** As used herein, "authentication" will refer broadly to the overall process of verifying someone's identity to be who he says he is. An "authentication technique" will refer to a particular type of authentication based upon a particular piece of knowledge, physical token, or biometric reading. "Authentication data" refers to information which is sent to or otherwise demonstrated to an authentication authority in order to establish identity. "Enrollment data" will refer to the data which is initially submitted to an authentication authority in order to establish a baseline for comparison with authentication data. An "authentication instance" will refer to the data associated with an attempt to authenticate by an authentication technique.

**[0198]** The internal protocols and communications involved in the process of authenticating a user is described with reference to FIGURE 10 above. The part of

this process within which the context sensitive authentication takes place occurs within the comparison step shown as step 1045 of FIGURE 10. This step takes place within the authentication engine 215 and involves assembling the enrollment data 410 retrieved from the depository 210 and comparing the authentication data provided by the user to it. One particular embodiment of this process is shown in FIGURE 16 and described below.

**[0199]** The current authentication data provided by the user and the enrollment data retrieved from the depository 210 are received by the authentication engine 215 in step 1600 of FIGURE 16. Both of these sets of data may contain data which is related to separate techniques of authentication. The authentication engine 215 separates the authentication data associated with each individual authentication instance in step 1605. This is necessary so that the authentication data is compared with the appropriate subset of the enrollment data for the user (e.g. fingerprint authentication data should be compared with fingerprint enrollment data, rather than password enrollment data).

**[0200]** Generally, authenticating a user involves one or more individual authentication instances, depending on which authentication techniques are available to the user. These methods are limited by the enrollment data which were provided by the user during his enrollment process (if the user did not provide a retinal scan when enrolling, he will not be able to authenticate himself using a retinal scan), as well as the means which may be currently available to the user (e.g. if the user does

not have a fingerprint reader at his current location, fingerprint authentication will not be practical). In some cases, a single authentication instance may be sufficient to authenticate a user; however, in certain circumstances a combination of multiple authentication instances may be used in order to more confidently authenticate a user for a particular transaction.

**[0201]** Each authentication instance consists of data related to a particular authentication technique (e.g. fingerprint, password, smart card, etc.) and the circumstances which surround the capture and delivery of the data for that particular technique. For example, a particular instance of attempting to authenticate via password will generate not only the data related to the password itself, but also circumstantial data, known as "metadata", related to that password attempt. This circumstantial data includes information such as: the time at which the particular authentication instance took place, the network address from which the authentication information was delivered, as well as any other information as is known to those of skill in the art which may be determined about the origin of the authentication data (the type of connection, the processor serial number, etc.).

**[0202]** In many cases, only a small amount of circumstantial metadata will be available. For example, if the user is located on a network which uses proxies or network address translation or another technique which masks the address of the originating computer, only the address of the proxy or router may be determined.

Similarly, in many cases information such as the processor serial number will not be available because of either limitations of the hardware or operating system being used, disabling of such features by the operator of the system, or other limitations of the connection between the user's system and the trust engine 110.

**[0203]** As shown in FIGURE 16, once the individual authentication instances represented within the authentication data are extracted and separated in step 1605, the authentication engine 215 evaluates each instance for its reliability in indicating that the user is who he claims to be. The reliability for a single authentication instance will generally be determined based on several factors. These may be grouped as factors relating to the reliability associated with the authentication technique, which are evaluated in step 1610, and factors relating to the reliability of the particular authentication data provided, which are evaluated in step 1815. The first group includes without limitation the inherent reliability of the authentication technique being used, and the reliability of the enrollment data being used with that method. The second group includes without limitation the degree of match between the enrollment data and the data provided with the authentication instance, and the metadata associated with that authentication instance. Each of these factors may vary independently of the others.

**[0204]** The inherent reliability of an authentication technique is based on how hard it is for an imposter to provide someone else's correct data, as well as the

overall error rates for the authentication technique. For passwords and knowledge based authentication methods, this reliability is often fairly low because there is nothing that prevents someone from revealing their password to another person and for that second person to use that password. Even a more complex knowledge based system may have only moderate reliability since knowledge may be transferred from person to person fairly easily. Token based authentication, such as having a proper smart card or using a particular terminal to perform the authentication, is similarly of low reliability used by itself, since there is no guarantee that the right person is in possession of the proper token.

**[0205]** However, biometric techniques are more inherently reliable because it is generally difficult to provide someone else with the ability to use your fingerprints in a convenient manner, even intentionally. Because subverting biometric authentication techniques is more difficult, the inherent reliability of biometric methods is generally higher than that of purely knowledge or token based authentication techniques. However, even biometric techniques may have some occasions in which a false acceptance or false rejection is generated. These occurrences may be reflected by differing reliabilities for different implementations of the same biometric technique. For example, a fingerprint matching system provided by one company may provide a higher reliability than one provided by a different company because one uses higher quality optics or a better scanning resolution or

some other improvement which reduces the occurrence of false acceptances or false rejections.

**[0206]** Note that this reliability may be expressed in different manners. The reliability is desirably expressed in some metric which can be used by the heuristics 530 and algorithms of the authentication engine 215 to calculate the confidence level of each authentication. One preferred mode of expressing these reliabilities is as a percentage or fraction. For instance, fingerprints might be assigned an inherent reliability of 97%, while passwords might only be assigned an inherent reliability of 50%. Those of skill in the art will recognize that these particular values are merely exemplary and may vary between specific implementations.

**[0207]** The second factor for which reliability must be assessed is the reliability of the enrollment. This is part of the "graded enrollment" process referred to above. This reliability factor reflects the reliability of the identification provided during the initial enrollment process. For instance, if the individual initially enrolls in a manner where they physically produce evidence of their identity to a notary or other public official, and enrollment data is recorded at that time and notarized, the data will be more reliable than data which is provided over a network during enrollment and only vouched for by a digital signature or other information which is not truly tied to the individual.

**[0208]** Other enrollment techniques with varying levels of reliability include without limitation: enrollment at

a physical office of the trust engine 110 operator; enrollment at a user's place of employment; enrollment at a post office or passport office; enrollment through an affiliated or trusted party to the trust engine 110 operator; anonymous or pseudonymous enrollment in which the enrolled identity is not yet identified with a particular real individual, as well as such other means as are known in the art.

**[0209]** These factors reflect the trust between the trust engine 110 and the source of identification provided during the enrollment process. For instance, if enrollment is performed in association with an employer during the initial process of providing evidence of identity, this information may be considered extremely reliable for purposes within the company, but may be trusted to a lesser degree by a government agency, or by a competitor. Therefore, trust engines operated by each of these other organizations may assign different levels of reliability to this enrollment.

**[0210]** Similarly, additional data which is submitted across a network, but which is authenticated by other trusted data provided during a previous enrollment with the same trust engine 110 may be considered as reliable as the original enrollment data was, even though the latter data were submitted across an open network. In such circumstances, a subsequent notarization will effectively increase the level of reliability associated with the original enrollment data. In this way for example, an anonymous or pseudonymous enrollment may then be raised to a full enrollment by demonstrating to some

enrollment official the identity of the individual matching the enrolled data.

**[0211]** The reliability factors discussed above are generally values which may be determined in advance of any particular authentication instance. This is because they are based upon the enrollment and the technique, rather than the actual authentication. In one embodiment, the step of generating reliability based upon these factors involves looking up previously determined values for this particular authentication technique and the enrollment data of the user. In a further aspect of an advantageous embodiment of the present invention, such reliabilities may be included with the enrollment data itself. In this way, these factors are automatically delivered to the authentication engine 215 along with the enrollment data sent from the depository 210.

**[0212]** While these factors may generally be determined in advance of any individual authentication instance, they still have an effect on each authentication instance which uses that particular technique of authentication for that user. Furthermore, although the values may change over time (e.g. if the user re-enrolls in a more reliable fashion), they are not dependent on the authentication data itself. By contrast, the reliability factors associated with a single specific instance's data may vary on each occasion. These factors, as discussed below, must be evaluated for each new authentication in order to generate reliability scores in step 1815.

**[0213]** The reliability of the authentication data reflects the match between the data provided by the user

in a particular authentication instance and the data provided during the authentication enrollment. This is the fundamental question of whether the authentication data matches the enrollment data for the individual the user is claiming to be. Normally, when the data do not match, the user is considered to not be successfully authenticated, and the authentication fails. The manner in which this is evaluated may change depending on the authentication technique used. The comparison of such data is performed by the comparator 515 function of the authentication engine 215 as shown in FIGURE 5.

**[0214]** For instance, matches of passwords are generally evaluated in a binary fashion. In other words, a password is either a perfect match, or a failed match. It is usually not desirable to accept as even a partial match a password which is close to the correct password if it is not exactly correct. Therefore, when evaluating a password authentication, the reliability of the authentication returned by the comparator 515 is typically either 100% (correct) or 0% (wrong), with no possibility of intermediate values.

**[0215]** Similar rules to those for passwords are generally applied to token based authentication methods, such as smart cards. This is because having a smart card which has a similar identifier or which is similar to the correct one, is still just as wrong as having any other incorrect token. Therefore tokens tend also to be binary authenticators: a user either has the right token, or he doesn't.

[0216] However, certain types of authentication data, such as questionnaires and biometrics, are generally not binary authenticators. For example, a fingerprint may match a reference fingerprint to varying degrees. To some extent, this may be due to variations in the quality of the data captured either during the initial enrollment or in subsequent authentications. (A fingerprint may be smudged or a person may have a still healing scar or burn on a particular finger.) In other instances the data may match less than perfectly because the information itself is somewhat variable and based upon pattern matching. (A voice analysis may seem close but not quite right because of background noise, or the acoustics of the environment in which the voice is recorded, or because the person has a cold.) Finally, in situations where large amounts of data are being compared, it may simply be the case that much of the data matches well, but some doesn't. (A ten-question questionnaire may have resulted in eight correct answers to personal questions, but two incorrect answers.) For any of these reasons, the match between the enrollment data and the data for a particular authentication instance may be desirably assigned a partial match value by the comparator 515. In this way, the fingerprint might be said to be a 85% match, the voice print a 65% match, and the questionnaire an 80% match, for example.

[0217] This measure (degree of match) produced by the comparator 515 is the factor representing the basic issue of whether an authentication is correct or not. However, as discussed above, this is only one of the factors which

may be used in determining the reliability of a given authentication instance. Note also that even though a match to some partial degree may be determined, that ultimately, it may be desirable to provide a binary result based upon a partial match. In an alternate mode of operation, it is also possible to treat partial matches as binary, i.e. either perfect (100%) or failed (0%) matches, based upon whether or not the degree of match passes a particular threshold level of match. Such a process may be used to provide a simple pass/fail level of matching for systems which would otherwise produce partial matches.

**[0218]** Another factor to be considered in evaluating the reliability of a given authentication instance concerns the circumstances under which the authentication data for this particular instance are provided. As discussed above, the circumstances refer to the metadata associated with a particular authentication instance. This may include without limitation such information as: the network address of the authenticator, to the extent that it can be determined; the time of the authentication; the mode of transmission of the authentication data (phone line, cellular, network, etc.); and the serial number of the system of the authenticator.

**[0219]** These factors can be used to produce a profile of the type of authentication that is normally requested by the user. Then, this information can be used to assess reliability in at least two manners. One manner is to consider whether the user is requesting

authentication in a manner which is consistent with the normal profile of authentication by this user. If the user normally makes authentication requests from one network address during business days (when she is at work) and from a different network address during evenings or weekends (when she is at home), an authentication which occurs from the home address during the business day is less reliable because it is outside the normal authentication profile. Similarly, if the user normally authenticates using a fingerprint biometric and in the evenings, an authentication which originates during the day using only a password is less reliable.

**[0220]** An additional way in which the circumstantial metadata can be used to evaluate the reliability of an instance of authentication is to determine how much corroboration the circumstance provides that the authenticator is the individual he claims to be. For instance, if the authentication comes from a system with a serial number known to be associated with the user, this is a good circumstantial indicator that the user is who they claim to be. Conversely, if the authentication is coming from a network address which is known to be in Los Angeles when the user is known to reside in London, this is an indication that this authentication is less reliable based on its circumstances.

**[0221]** It is also possible that a cookie or other electronic data may be placed upon the system being used by a user when they interact with a vendor system or with the trust engine 110. This data is written to the storage of the system of the user and may contain an

identification which may be read by a Web browser or other software on the user system. If this data is allowed to reside on the user system between sessions (a "persistent cookie"), it may be sent with the authentication data as further evidence of the past use of this system during authentication of a particular user. In effect, the metadata of a given instance, particularly a persistent cookie, may form a sort of token based authenticator itself.

**[0222]** Once the appropriate reliability factors based on the technique and data of the authentication instance are generated as described above in steps 1610 and 1615 respectively, they are used to produce an overall reliability for the authentication instance provided in step 1620. One means of doing this is simply to express each reliability as a percentage and then to multiply them together.

**[0223]** For example, suppose the authentication data is being sent in from a network address known to be the user's home computer completely in accordance with the user's past authentication profile (100%), and the technique being used is fingerprint identification (97%), and the initial finger print data was roistered through the user's employer with the trust engine 110 (90%), and the match between the authentication data and the original fingerprint template in the enrollment data is very good (99%). The overall reliability of this authentication instance could then be calculated as the product of these reliabilities: 100% * 97% * 90% * 99% - 86.4% reliability.

[0224] This calculated reliability represents the reliability of one single instance of authentication. The overall reliability of a single authentication instance may also be calculated using techniques which treat the different reliability factors differently, for example by using formulas where different weights are assigned to each reliability factor. Furthermore, those of skill in the art will recognize that the actual values used may represent values other than percentages and may use non-arithmetic systems. One embodiment may include a module used by an authentication requestor to set the weights for each factor and the algorithms used in establishing the overall reliability of the authentication instance.

[0225] The authentication engine 215 may use the above techniques and variations thereof to determine the reliability of a single authentication instance, indicated as step 1620. However, it may be useful in many authentication situations for multiple authentication instances to be provided at the same time. For example, while attempting to authenticate himself using the system of the present invention, a user may provide a user identification, fingerprint authentication data, a smart card, and a password. In such a case, three independent authentication instances are being provided to the trust engine 110 for evaluation. Proceeding to step 1625, if the authentication engine 215 determines that the data provided by the user includes more than one authentication instance, then each instance in turn will be selected as shown in step 1630 and

evaluated as described above in steps 1610, 1615 and 1620.

**[0226]** Note that many of the reliability factors discussed may vary from one of these instances to another. For instance, the inherent reliability of these techniques is likely to be different, as well as the degree of match provided between the authentication data and the enrollment data. Furthermore, the user may have provided enrollment data at different times and under different circumstances for each of these techniques, providing different enrollment reliabilities for each of these instances as well. Finally, even though the circumstances under which the data for each of these instances is being submitted is the same, the use of such techniques may each fit the profile of the user differently, and so may be assigned different circumstantial reliabilities. (For example, the user may normally use their password and fingerprint, but not their smart card.)

**[0227]** As a result, the final reliability for each of these authentication instances may be different from One another. However, by using multiple instances together, the overall confidence level for the authentication will tend to increase.

**[0228]** Once the authentication engine has performed steps 1610 through 1620 for all of the authentication instances provided in the authentication data, the reliability of each instance is used in step 1635 to evaluate the overall authentication confidence level. This process of combining the individual authentication

instance reliabilities into the authentication confidence level may be modeled by various methods relating the individual reliabilities produced, and may also address the particular interaction between some of these authentication techniques. (For example, multiple knowledge-based systems such as passwords may produce less confidence than a single password and even a fairly weak biometric, such as a basic voice analysis.)

**[0229]** One means in which the authentication engine 215 may combine the reliabilities of multiple concurrent authentication instances to generate a final confidence level is to multiply the unreliability of each instance to arrive at a total unreliability. The unreliability is generally the complementary percentage of the reliability. For example, a technique which is 84% reliable is 16% unreliable. The three authentication instances described above (fingerprint, smart card, password) which produce reliabilities of 86%, 75%, and 72% would have corresponding unreliabilities of (100- 86)%, (100- 75)% and (100- 72)%, or 14%, 25%, and 28%, respectively. By multiplying these unreliabilities, we get a cumulative unreliability of 14% * 25% * 28% - .98% unreliability, which corresponds to a reliability of 99.02%.

**[0230]** In an additional mode of operation, additional factors and heuristics 530 may be applied within the authentication engine 215 to account for the interdependence of various authentication techniques. For example, if someone has unauthorized access to a particular home computer, they probably have access to

the phone line at that address as well. Therefore, authenticating based on an originating phone number as well as upon the serial number of the authenticating system does not add much to the overall confidence in the authentication. However, knowledge based authentication is largely independent of token based authentication (i.e. if someone steals your cellular phone or keys, they are no more likely to know your PIN or password than if they hadn't).

**[0231]** Furthermore, different vendors or other authentication requestors may wish to weigh different aspects of the authentication differently. This may include the use of separate weighing factors or algorithms used in calculating the reliability of individual instances as well as the use of different means to evaluate authentication events with multiple instances.

**[0232]** For instance, vendors for certain types of transactions, for instance corporate email systems, may desire to authenticate primarily based upon heuristics and other circumstantial data by default. Therefore, they may apply high weights to factors related to the metadata and other profile related information associated with the circumstances surrounding authentication events. This arrangement could be used to ease the burden on users during normal operating hours, by not requiring more from the user than that he be logged on to the correct machine during business hours. However, another vendor may weigh authentications coming from a particular technique most heavily, for instance fingerprint

matching, because of a policy decision that such a technique is most suited to authentication for the particular vendor's purposes.

[0233] Such varying weights may be defined by the authentication requestor in generating the authentication request and sent to the trust engine 110 with the authentication request in one mode of operation. Such options could also be set as preferences during an initial enrollment process for the authentication requestor and stored within the authentication engine in another mode of operation.

[0234] Once the authentication engine 215 produces an authentication confidence level for the authentication data provided, this confidence level is used to complete the authentication request in step 1640, and this information is forwarded from the authentication engine 215 to the transaction engine 205 for inclusion in a message to the authentication requestor.

[0235] The process described above is merely exemplary, and those of skill in the art will recognize that the steps need not be performed in the order shown or that only certain of the steps are desired to be performed, or that a variety of combinations of steps may be desired. Furthermore, certain steps, such as the evaluation of the reliability of each authentication instance provided, may be carried out in parallel with one another if circumstances permit.

[0236] In a further aspect of this invention, a method is provided to accommodate conditions when the authentication confidence level produced by the process

described above fails to meet the required trust level of the vendor or other party requiring the authentication. In circumstances such as these where a gap exists between the level of confidence provided and the level of trust desired, the operator of the trust engine 110 is in a position to provide opportunities for one or both parties to provide alternate data or requirements in order to close this trust gap. This process will be referred to as "trust arbitrage" herein.

**[0237]** Trust arbitrage may take place within a framework of cryptographic authentication as described above with reference to FIGURES 10 and 11. As shown therein, a vendor or other party will request authentication of a particular user in association with a particular transaction. In one circumstance, the vendor simply requests an authentication, either positive or negative, and after receiving appropriate data from the user, the trust engine 110 will provide such a binary authentication. In circumstances such as these, the degree of confidence required in order to secure a positive authentication is determined based upon preferences set within the trust engine 110.

**[0238]** However, it is also possible that the vendor may request a particular level of trust in order to complete a particular transaction. This required level may be included with the authentication request (e.g. authenticate this user to 98% confidence) or may be determined by the trust engine 110 based on other factors associated with the transaction (i.e. authenticate this user as appropriate for this transaction). One such

factor might be the economic value of the transaction. For transactions which have greater economic value, a higher degree of trust may be required. Similarly, for transactions with high degrees of risk a high degree of trust may be required. Conversely, for transactions which are either of low risk or of low value, lower trust levels may be required by the vendor or other authentication requestor.

**[0239]** The process of trust arbitrage occurs between the steps of the trust engine 110 receiving the authentication data in step 1050 of FIGURE 10 and the return of an authentication result to the vendor in step 1055 of FIGURE 10. Between these steps, the process which leads to the evaluation of trust levels and the potential trust arbitrage occurs as shown in FIGURE 17. In circumstances where simple binary authentication is performed, the process shown in FIGURE 17 reduces to having the transaction engine 205 directly compare the authentication data provided with the enrollment data for the identified user as discussed above with reference to FIGURE 10, flagging any difference as a negative authentication.

**[0240]** As shown in FIGURE 17, the first step after receiving the data in step 1050 is for the transaction engine 205 to determine the trust level which is required for a positive authentication for this particular transaction in step 1710. This step may be performed by one of several different methods. The required trust level may be specified to the trust engine 110 by the authentication requestor at the time when the

authentication request is made. The authentication requestor may also set a preference in advance which is stored within the depository 210 or other storage which is accessible by the transaction engine 205. This preference may then be read and used each time an authentication request is made by this authentication requestor. The preference may also be associated with a particular user as a security measure such that a particular level of trust is always required in order to authenticate that user, the user preference being stored in the depository 210 or other storage media accessible by the transaction engine 205. The required level may also be derived by the transaction engine 205 or authentication engine 215 based upon information provided in the authentication request, such as the value and risk level of the transaction to be authenticated.

**[0241]** In one mode of operation, a policy management module or other software which is used when generating the authentication request is used to specify the required degree of trust for the authentication of the transaction. This may be used to provide a series of rules to follow when assigning the required level of trust based upon the policies which are specified within the policy management module. One advantageous mode of operation is for such a module to be incorporated with the web server of a vendor in order to appropriately determine required level of trust for transactions initiated with the vendor's web server. In this way, transaction requests from users may be assigned a required trust level in accordance with the policies of

the vendor and such information may be forwarded to the trust engine 110 along with the authentication request.

**[0242]** This required trust level correlates with the degree of certainty that the vendor wants to have that the individual authenticating is in fact who he identifies himself as. For example, if the transaction is one where the vendor wants a fair degree of certainty because goods are changing hands, the vendor may require a trust level of 85%. For situation where the vendor is merely authenticating the user to allow him to view members only content or exercise privileges on a chat room, the downside risk may be small enough that the vendor requires only a 60% trust level. However, to enter into a production contract with a value of tens of thousands of dollars, the vendor may require a trust level of 99% or more.

**[0243]** This required trust level represents a metric to which the user must authenticate himself in order to complete the transaction. If the required trust level is 85% for example, the user must provide authentication to the trust engine 110 sufficient for the trust engine 110 to say with 85% confidence that the user is who they say they are. It is the balance between this required trust level and the authentication confidence level which produces either a positive authentication (to the satisfaction of the vendor) or a possibility of trust arbitrage.

**[0244]** As shown in FIGURE 17, after the transaction engine 205 receives the required trust level, it compares in step 1720 the required trust level to the

authentication confidence level which the authentication engine 215 calculated for the current authentication (as discussed with reference to FIGURE 16). If the authentication confidence level is higher than the required trust level for the transaction in step 1730, then the process moves to step 1740 where a positive authentication for this transaction is produced by the transaction engine 205. A message to this effect will then be inserted into the authentication results and returned to the vendor by the transaction engine 205 as shown in step 1055 (see FIGURE 10).

**[0245]** However, if the authentication confidence level does not fulfill the required trust level in step 1730, then a confidence gap exists for the current authentication, and trust arbitrage is conducted in step 1750. Trust arbitrage is described more completely with reference to FIGURE 18 below. This process as described below takes place within the transaction engine 205 of the trust engine 110. Because no authentication or other cryptographic operations are needed to execute trust arbitrage (other than those required for the SSL communication between the transaction engine 205 and other components), the process may be performed outside the authentication engine 215. However, as will be discussed below, any reevaluation of authentication data or other cryptographic or authentication events will require the transaction engine 205 to resubmit the appropriate data to the authentication engine 215. Those of skill in the art will recognize that the trust arbitrage process could alternately be structured to take

place partially or entirely within the authentication engine 215 itself.

**[0246]** As mentioned above, trust arbitrage is a process where the trust engine 110 mediates a negotiation between the vendor and user in an attempt to secure a positive authentication where appropriate. As shown in step 1805, the transaction engine 205 first determines whether or not the current situation is appropriate for trust arbitrage. This may be determined based upon the circumstances of the authentication, e.g. whether this authentication has already been through multiple cycles of arbitrage, as well as upon the preferences of either the vendor or user, as will be discussed further below.

**[0247]** In such circumstances where arbitrage is not possible, the process proceeds to step 1810 where the transaction engine 205 generates a negative authentication and then inserts it into the authentication results which are sent to the vendor in step 1055 (see FIGURE 10). One limit which may be advantageously used to prevent authentications from pending indefinitely is to set a time-out period from the initial authentication request. In this way, any transaction which is not positively authenticated within the time limit is denied further arbitrage and negatively authenticated. Those of skill in the art will recognize that such a time limit may vary depending upon the circumstances of the transaction and the desires of the user and vendor. Limitations may also be placed upon the number of attempts that may be made at providing a

successful authentication. Such limitations may be handled by an attempt limiter 535 as shown in FIGURE 5.

**[0248]** If arbitrage is not prohibited in step 1805, the transaction engine 205 will then engage in negotiation with one or both of the transacting parties. The transaction engine 205 may send a message to the user requesting some form of additional authentication in order to boost the authentication confidence level produced as shown in step 1820. In the simplest form, this may simply indicates that authentication was insufficient. A request to produce one or more additional authentication instances to improve the overall confidence level of the authentication may also be sent.

**[0249]** If the user provides some additional authentication instances in step 1825, then the transaction engine 205 adds these authentication instances to the authentication data for the transaction and forwards it to the authentication engine 215 as shown in step 1015 (see FIGURE 10), and the authentication is reevaluated based upon both the pre-existing authentication instances for this transaction and the newly provided authentication instances.

**[0250]** An additional type of authentication may be a request from the trust engine 110 to make some form of person-to-person contact between the trust engine 110 operator (or a trusted associate) and the user, for example, by phone call. This phone call or other non-computer authentication can be used to provide personal contact with the individual and also to conduct

some form of questionnaire based authentication. This also may give the opportunity to verify an originating telephone number and potentially a voice analysis of the user when he calls in. Even if no additional authentication data can be provided, the additional context associated with the user's phone number may improve the reliability of the authentication context. Any revised data or circumstances based upon this phone call are fed into the trust engine 110 for use in consideration of the authentication request.

**[0251]** Additionally, in step 1820 the trust engine 110 may provide an opportunity for the user to purchase insurance, effectively buying a more confident authentication. The operator of the trust engine 110 may, at times, only want to make such an option available if the confidence level of the authentication is above a certain threshold to begin with. In effect, this user side insurance is a way for the trust engine 110 to vouch for the user when the authentication meets the normal required trust level of the trust engine 110 for authentication, but does not meet the required trust level of the vendor for this transaction. In this way, the user may still successfully authenticate to a very high level as may be required by the vendor, even though he only has authentication instances which produce confidence sufficient for the trust engine 110.

**[0252]** This function of the trust engine 110 allows the trust engine 110 to vouch for someone who is authenticated to the satisfaction of the trust engine 110, but not of the vendor. This is analogous to the

function performed by a notary in adding his signature to a document in order to indicate to someone reading the document at a later time that the person whose signature appears on the document is in fact the person who signed it. The signature of the notary testifies to the act of signing by the user. In the same way, the trust engine is providing an indication that the person transacting is who they say they are.

**[0253]** However, because the trust engine 110 is artificially boosting the level of confidence provided by the user, there is a greater risk to the trust engine 110 operator, since the user is not actually meeting the required trust level of the vendor. The cost of the insurance is designed to offset the risk of a false positive authentication to the trust engine 110 (who may be effectively notarizing the authentications of the user). The user pays the trust engine 110 operator to take the risk of authenticating to a higher level of confidence than has actually been provided.

**[0254]** Because such an insurance system allows someone to effectively buy a higher confidence rating from the trust engine 110, both vendors and users may wish to prevent the use of user side insurance in certain transactions. Vendors may wish to limit positive authentications to circumstances where they know that actual authentication data supports the degree of confidence which they require and so may indicate to the trust engine 110 that user side insurance is not to be allowed. Similarly, to protect his online identity, a user may wish to prevent the use of user side insurance

on his account, or may wish to limit its use to situations where the authentication confidence level without the insurance is higher than a certain limit. This may be used as a security measure to prevent someone from overhearing a password or stealing a smart card and using them to falsely authenticate to a low level of confidence, and then purchasing insurance to produce a very high level of (false) confidence. These factors may be evaluated in determining whether user side insurance is allowed.

**[0255]** If user purchases insurance in step 1840, then the authentication confidence level is adjusted based upon the insurance purchased in step 1845, and the authentication confidence level and required trust level are again compared in step 1730 (see FIGURE 17). The process continues from there, and may lead to either a positive authentication in step 1740 (see FIGURE 17), or back into the trust arbitrage process in step 1750 for either further arbitrage (if allowed) or a negative authentication in step 1810 if further arbitrage is prohibited.

**[0256]** In addition to sending a message to the user in step 1820, the transaction engine 205 may also send a message to the vendor in step 1830 which indicates that a pending authentication is currently below the required trust level. The message may also offer various options on how to proceed to the vendor. One of these Options is to simply inform the vendor of what the current authentication confidence level is and ask if the vendor wishes to maintain their current unfulfilled required

trust level. This may be beneficial because in some cases, the vendor may have independent means for authenticating the transaction or may have been using a default set of requirements which generally result in a higher required level being initially specified than is actually needed for the particular transaction at hand.

**[0257]** For instance, it may be standard practice that all incoming purchase order transactions with the vendor are expected to meet a 98% trust level. However, if an order was recently discussed by phone between the vendor and a long-standing customer, and immediately thereafter the transaction is authenticated, but only to a 93% confidence level, the vendor may wish to simply lower the acceptance threshold for this transaction, because the phone call effectively provides additional authentication to the vendor. In certain circumstances, the vendor may be willing to lower their required trust level, but not all the way to the level of the current authentication confidence. For instance, the vendor in the above example might consider that the phone call prior to the order might merit a 4% reduction in the degree of trust needed; however, this is still greater than the 93% confidence produced by the user.

**[0258]** If the vendor does adjust their required trust level in step 1835, then the authentication confidence level produced by the authentication and the required trust level are compared in step 1730 (see FIGURE 17). If the confidence level now exceeds the required trust level, a positive authentication may be generated in the transaction engine 205 in step 1740 (see FIGURE 17). If

not, further arbitrage may be attempted as discussed above if it is permitted.

**[0259]** In addition to requesting an adjustment to the required trust level, the transaction engine 205 may also offer vendor side insurance to the vendor requesting the authentication. This insurance serves a similar purpose to that described above for the user side insurance. Here, however, rather than the cost corresponding to the risk being taken by the trust engine 110 in authenticating above the actual authentication confidence level produced, the cost of the insurance corresponds to the risk being taken by the vendor in accepting a lower trust level in the authentication.

**[0260]** Instead of just lowering their actual required trust level, the vendor has the option of purchasing insurance to protect itself from the additional risk associated with a lower level of trust in the authentication of the user. As described above, it may be advantageous for the vendor to only consider purchasing such insurance to cover the trust gap in conditions where the existing authentication is already above a certain threshold.

**[0261]** The availability of such vendor side insurance allows the vendor the option to either: lower his trust requirement directly at no additional cost to himself, bearing the risk of a false authentication himself (based on the lower trust level required); or, buying insurance for the trust gap between the authentication confidence level and his requirement, with the trust engine 110 operator bearing the risk of the lower confidence level

which has been provided. By purchasing the insurance, the vendor effectively keeps his high trust level requirement; because the risk of a false authentication is shifted to the trust engine 110 operator.

**[0262]** If the vendor purchases insurance in step 1840, the authentication confidence level and required trust levels are compared in step 1730 (see FIGURE 17), and the process continues as described above.

**[0263]** Note that it is also possible that both the user and the vendor respond to messages from the trust engine 110. Those of skill in the art will recognize that there are multiple ways in which such situations can be handled. One advantageous mode of handling the possibility of multiple responses is simply to treat the responses in a first-come, first-served manner. For example, if the vendor responds with a lowered required trust level and immediately thereafter the user also purchases insurance to raise his authentication level, the authentication is first reevaluated based upon the lowered trust requirement from the vendor. If the authentication is now positive, the user's insurance purchase is ignored. In another advantageous mode of operation, the user might only be charged for the level of insurance required to meet the new, lowered trust requirement of the vendor (if a trust gap remained even with the lowered vendor trust requirement).

**[0264]** If no response from either party is received during the trust arbitrage process at step 1850 within the time limit set for the authentication, the arbitrage is reevaluated in step 1805. This effectively begins the

arbitrage process again. If the time limit was final or other circumstances prevent further arbitrage in step 1805, a negative authentication is generated by the transaction engine 205 in step 1810 and returned to the vendor in step 1055 (see FIGURE 10). If not, new messages may be sent to the user and vendor, and the process may be repeated as desired.

**[0265]** Note that for certain types of transactions, for instance, digitally signing documents which are not part of a transaction, there may not necessarily be a vendor or other third party; therefore the transaction is primarily between the user and the trust engine 110. In circumstances such as these, the trust engine 110 will have its own required trust level which must be satisfied in order to generate a positive authentication. However, in such circumstances, it will often not be desirable for the trust engine 110 to offer insurance to the user in order for him to raise the confidence of his own signature.

**[0266]** The process described above and shown in FIGURES 16-18 may be carried out using various communications modes as described above with reference to the trust engine 110. For instance, the messages may be web-based and sent using SSL connections between the trust engine 110 and applets downloaded in real time to browsers running on the user or vendor systems. In an alternate mode of operation, certain dedicated applications may be in use by the user and vendor which facilitate such arbitrage and insurance transactions. In another alternate mode of operation, secure email

operations may be used to mediate the arbitrage described above, thereby allowing deferred evaluations and batch processing of authentications. Those of skill in the art will recognize that different communications modes may be used as are appropriate for the circumstances and authentication requirements of the vendor.

**[0267]** The following description with reference to FIGURE 19 describes a sample transaction which integrates the various aspects of the present invention as described above. This example illustrates the overall process between a user and a vendor as mediates by the trust engine 110. Although the various steps and components as described in detail above may be used to carry out the following transaction, the process illustrated focuses on the interaction between the trust engine 110, user and vendor.

**[0268]** The transaction begins when the user, while viewing web pages online, fills out an order form on the web site of the vendor in step 1900. The user wishes to submit this order form to the vendor, signed with his digital signature. In order to do this, the user submits the order form with his request for a signature to the trust engine 110 in step 1905. The user will also provide authentication data which will be used as described above to authenticate his identity.

**[0269]** In step 1910 the authentication data is compared to the enrollment data by the trust engine 110 as discussed above, and if a positive authentication is produced, the hash of the order form, signed with the

private key of the user, is forwarded to the vendor along with the order form itself.

**[0270]** The vendor receives the signed form in step 1915, and then the vendor will generate an invoice or other contract related to the purchase to be made in step 1920. This contract is sent back to the user with a request for a signature in step 1925. The vendor also sends an authentication request for this contract transaction to the trust engine 110 in step 1930 including a hash of the contract which will be signed by both parties. To allow the contract to be digitally signed by both parties, the vendor also includes authentication data for itself so that the vendor's signature upon the contract can later be verified if necessary.

**[0271]** As discussed above, the trust engine 110 then verifies the authentication data provided by the vendor to confirm the vendor's identity, and if the data produces a positive authentication in step 1935, continues with step 1955 when the data is received from the user. If the vendor's authentication data does not match the enrollment data of the vendor to the desired degree, a message is returned to the vendor requesting further authentication. Trust arbitrage may be performed here if necessary, as described above, in order for the vendor to successfully authenticate itself to the trust engine 110.

**[0272]** When the user receives the contract in step 1940, he reviews it, generates authentication data to sign it if it is acceptable in step 1945, and then sends

a hash of the contract and his authentication data to the trust engine 110 in step 1950. The trust engine 110 verifies the authentication data in step 1955 and if the authentication is good, proceeds to process the contract as described below. As discussed above with reference to FIGURES 17 and 18, trust arbitrage may be performed as appropriate to close any trust gap which exists between the authentication confidence level and the required authentication level for the transaction.

**[0273]** The trust engine 110 signs the hash of the contract with the user's private key, and sends this signed hash to the vendor in step 1960, signing the complete message on its own behalf, i.e., including a hash of the complete message (including the user's signature) encrypted with the private key 510 of the trust engine 110. This message is received by the vendor in step 1965. The message represents a signed contract (hash of contract encrypted using user's private key) and a receipt from the trust engine 110 (the hash of the message including the signed contract, encrypted using the trust engine 110's private key).

**[0274]** The trust engine 110 similarly prepares a hash of the contract with the vendor's private key in step 1970, and forwards this to the user, signed by the trust engine 110. In this way, the user also receives a copy of the contract, signed by the vendor, as well as a receipt, signed by the trust engine 110, for delivery of the signed contract in step 1975.

**[0275]** In addition to the foregoing, an additional aspect of the invention provides a cryptographic Service

Provider Module (SPM) which may be available to a client side application as a means to access functions provided by the trust engine 110 described above. One advantageous way to provide such a service is for the cryptographic SPM is to mediate communications between a third party Application Programming Interface (API) and a trust engine 110 which is accessible via a network or other remote connection. A sample cryptographic SPM is described below with reference to FIGURE 20.

**[0276]** For example, on a typical system, a number of API's are available to programmers. Each API provides a set of function calls which may be made by an application 2000 running upon the system. Examples of API's which provide programming interfaces suitable for cryptographic functions, authentication functions, and other security function include the Cryptographic API (CAPI) 2010 provided by Microsoft with its Windows operating systems, and the Common Data Security Architecture (CDSA), sponsored by IBM, Intel and other members of the Open Group. CAPI will be used as an exemplary security API in the discussion that follows. However, the cryptographic SPM described could be used with CDSA or other security API's as are known in the art.

**[0277]** This API is used by a user system 105 or vendor system 120 when a call is made for a cryptographic function. Included among these functions may be requests associated with performing various cryptographic operations, such as encrypting a document with a particular key, signing a document, requesting a digital certificate, verifying a signature upon a signed

document, and such other cryptographic functions as are described herein or known to those of skill in the art.

**[0278]** Such cryptographic functions are normally performed locally to the system upon which CAPI 2010 is located. This is because generally the functions called require the use of either resources of the local user system 105, such as a fingerprint reader, or software functions which are programmed using libraries which are executed on the local machine. Access to these local resources is normally provided by one or more Service Provider Modules (SPM's) 2015, 2020 as referred to above which provide resources with which the cryptographic functions are carried out. Such SPM's may include software libraries 2015 to perform encrypting or decrypting operations, or drivers and applications 2020 which are capable of accessing specialized hardware 2025, such as biometric scanning devices. In much the way that CAPI 2010 provides functions which may be used by an application 2000 of the system 105, the SPM's 2015, 2020 provide CAPI with access to the lower level functions and resources associated with the available services upon the system.

**[0279]** In accordance with the invention, it is possible to provide a cryptographic SPM 2030 which is capable of accessing the cryptographic functions provided by the trust engine 110 and making these functions available to an application 2000 through CAPI 2010. Unlike embodiments where CAPI 2010 is only able to access resources which are locally available through SPM's 2015, 2020, a cryptographic SPM 2030 as described herein would

be able to submit requests for cryptographic operations to a remotely-located, network-accessible trust engine 110 in order to perform the operations desired.

**[0280]** For instance, if an application 2000 has a need for a cryptographic operation, such as signing a document, the application 2000 makes a function call to the appropriate CAPI 2010 function. CAPI 2010 in turn will execute this function, making use of the resources which are made available to it by the SPM's 2015, 2020 and the cryptographic SPM 2030. In the case of a digital signature function, the cryptographic SPM 2030 will generate an appropriate request which will be sent to the trust engine 110 across the communication link 125.

**[0281]** The operations which occur between the cryptographic SPM 2030 and the trust engine 110 are the same operations that would be possible between any other system and the trust engine 110. However, these functions are effectively made available to a user system 105 through CAPI 2010 such that they appear to be locally available upon the user system 105 itself. However, unlike ordinary SPM's 2015, 2020, the functions are being carried out on the remote trust engine 110 and the results relayed to the cryptographic SPM 2030 in response to appropriate requests across the communication link 125.

**[0282]** This cryptographic SPM 2030 makes a number of operations available to the user system 105 or a vendor system 120 which might not otherwise be available. These functions include without limitation: encryption and decryption of documents; issuance of digital

certificates; digital signing of documents; verification of digital signatures; and such other operations as will be apparent to those of skill in the art.

[0283] In a separate embodiment, the present invention comprises a complete system for performing the data securing methods of the present invention on any data set. The computer system of this embodiment comprises a data splitting module that comprises the functionality shown in FIGURE 8 and described herein. In one embodiment of the present invention, the data splitting module, sometimes referred to herein as a secure data parser, comprises a parser program or software suite which comprises data splitting, encryption and decryption, reconstitution or reassembly functionality. This embodiment may further comprise a data storage facility or multiple data storage facilities, as well. The data splitting module, or secure data parser, comprises a cross-platform software module suite which integrates within an electronic infrastructure, or as an add-on to any application which requires the ultimate security of its data elements. This parsing process operates on any type of data set, and on any and all file types, or in a database on any row, column or cell of data in that database.

[0284] The parsing process of the present invention may, in one embodiment, be designed in a modular tiered fashion, and any encryption process is suitable for use in the process of the present invention. The modular tiers of the parsing and splitting process of the present invention may include, but are not limited to, 1)

cryptographic split, dispersed and securely stored in multiple locations; 2) encrypt, cryptographically split, dispersed and securely stored in multiple locations; 3) encrypt, cryptographically split, encrypt each share, then dispersed and securely stored in multiple locations; and 4) encrypt, cryptographically split, encrypt each share with a different type of encryption than was used in the first step, then dispersed and securely stored in multiple locations.

**[0285]** The process comprises, in one embodiment, splitting of the data according to the contents of a generated random number, or key and performing the same cryptographic splitting of the key used in the encryption of splitting of the data to be secured into two or more portions, or shares, of parsed and split data, and in one embodiment, preferably into four or more portions of parsed and split data, encrypting all of the portions, then scattering and storing these portions back into the database, or relocating them to any named device, fixed or removable, depending on the requestor's need for privacy and security. Alternatively, in another embodiment, encryption may occur prior to the splitting of the data set by the splitting module or secure data parser. The original data processed as described in this embodiment is encrypted and obfuscated and is secured. The dispersion of the encrypted elements, if desired, can be virtually anywhere, including, but not limited to, a single server or data storage device, or among separate data storage facilities or devices. Encryption key management in one embodiment may be included within the

software suite, or in another embodiment may be integrated into an existing infrastructure or any other desired location.

**[0286]** A cryptographic split (cryptosplit) partitions the data into N number of shares. The partitioning can be on any size unit of data, including an individual bit, bits, bytes, kilobytes, megabytes, or larger units, as well as any pattern or combination of data unit sizes whether predetermined or randomly generated. The units can also be of different sized, based on either a random or predetermined set of values. This means the data can be viewed as a sequence of these units. In this manner the size of the data units themselves may render the data more secure, for example by using one or more predetermined or randomly generated pattern, sequence or combination of data unit sizes. The units are then distributed (either randomly or by a predetermined set of values) into the N shares. This distribution could also involve a shuffling of the order of the units in the shares. It is readily apparent to those of ordinary skill in the art that the distribution of the data units into the shares may be performed according to a wide variety of possible selections, including but not limited to size-fixed, predetermined sizes, or one or more combination, pattern or sequence of data unit sizes that are predetermined or randomly generated.

**[0287]** One example of this cryptographic split process, or cryptosplit, would be to consider the data to be 23 bytes in size, with the data unit size chosen to be one byte, and with the number of shares selected to be 4.

Each byte would be distributed into one of the 4 shares. Assuming a random distribution, a key would be obtained to create a sequence of 23 random numbers (r1, r2, r3 through r23), each with a value between 1 and 4 corresponding to the four shares. Each of the units of data (in this example 23 individual bytes of data) is associated with one of the 23 random numbers corresponding to one of the four shares. The distribution of the bytes of data into the four shares would occur by placing the first byte of the data into share number r1, byte two into share r2, byte three into share r3, through the 23rd byte of data into share r23. It is readily apparent to those of ordinary skill in the art that a wide variety of other possible steps or combination or sequence of steps, including the size of the data units, may be used in the cryptosplit process of the present invention, and the above example is a non-limiting description of one process for cryptosplitting data. To recreate the original data, the reverse operation would be performed.

**[0288]** In another embodiment of the cryptosplit process of the present invention, an option for the cryptosplitting process is to provide sufficient redundancy in the shares such that only a subset of the shares are needed to reassemble or restore the data to its original or useable form. As a non-limiting example, the cryptosplit may be done as a "3 of 4" cryptosplit such that only three of the four shares are necessary to reassemble or restore the data to its original or useable form. This is also referred to as a "M of N cryptosplit"

wherein N is the total number of shares, and M is at least one less than N. It is readily apparent to those of ordinary skill in the art that there are many possibilities for creating this redundancy in the cryptosplitting process of the present invention.

**[0289]** In one embodiment of the cryptosplitting process of the present invention, each unit of data is stored in two shares, the primary share and the backup share. Using the "3 of 4" cryptosplitting process described above, any one share can be missing, and this is sufficient to reassemble or restore the original data with no missing data units since only three of the total four shares are required. As described herein, a random number is generated that corresponds to one of the shares. The random number is associated with a data unit, and stored in the corresponding share, based on a key. One key is used, in this embodiment, to generate the primary and backup share random number. As described herein for the cryptosplitting process of the present invention, a set of random numbers (also referred to as primary share numbers) from 0 to 3 are generated equal to the number of data units. Then another set of random numbers is generated (also referred to as backup share numbers) from 1 to 3 equal to the number of data units. Each unit of data is then associated with a primary share number and a backup share number. Alternatively, a set of random numbers may be generated that is fewer than the number of data units, and repeating the random number set, but this may reduce the security of the sensitive data. The primary share number

is used to determine into which share the data unit is stored. The backup share number is combined with the primary share number to create a third share number between 0 and 3, and this number is used to determine into which share the data unit is stored. In this example, the equation to determine the third share number is:

(primary share number + backup share number) MOD 4 = third share number.

**[0290]** In the embodiment described above where the primary share number is between 0 and 3, and the backup share number is between 1 and 3 ensures that the third share number is different from the primary share number. This results in the data unit being stored in two different shares. It is readily apparent to those of ordinary skill in the art that there are many ways of performing redundant cryptosplitting and non-redundant cryptosplitting in addition to the embodiments disclosed herein. For example, the data units in each share could be shuffled utilizing a different algorithm. This data unit shuffling may be performed as the original data is split into the data units, or after the data units are placed into the shares, or after the share is full, for example.

**[0291]** The various cryptosplitting processes and data shuffling processes described herein, and all other embodiments of the cryptosplitting and data shuffling methods of the present invention may be performed on data units of any size, including but not limited to, as small

as an individual bit, bits, bytes, kilobytes, megabytes or larger.

[0292] An example of one embodiment of source code that would perform the cryptosplitting process described herein is:

```
DATA [1:24] - array of bytes with the data to be split
SHARES[0:3; 1:24] - 2-dimensionalarray with each row
representing one of the shares
RANDOM[1:24] - array random numbers in the range of 0..3
S1 = 1;
S2 = 1;
S3 = 1;
S4 = 1;

For J = 1 to 24 do
     Begin
     IF RANDOM[J[ ==0 then
          Begin
          SHARES[1,S1] = DATA [J];
          S1 = S1 + 1;
          End
     ELSE IF RANDOM[J[ ==1 then
          Begin
          SHARES[2,S2] = DATA [J];
          S2 = S2 + 1;
          END
     ELSE IF RANDOM[J[ ==2 then
          Begin
          Shares[3,S3] = data [J];
          S3 = S3 + 1;
```

```
            End
        Else   begin
            Shares[4,S4] = data [J];
            S4 = S4 + 1;
            End;
        END;
```

**[0293]** An example of one embodiment of source code that would perform the cryptosplitting RAID process described herein is:

**[0294]** Generate two sets of numbers, PrimaryShare is 0 to 3, BackupShare is 1 to 3. Then put each data unit into share[primaryshare[1]] and share[(primaryshare[1]+backupshare[1]) mod 4, with the same process as in cryptosplitting described above. This method will be scalable to any size N, where only N-1 shares are necessary to restore the data.

**[0295]** The retrieval, recombining, reassembly or reconstituting of the encrypted data elements may utilize any number of authentication techniques, including, but not limited to, biometrics, such as fingerprint recognition, facial scan, hand scan, iris scan, retinal scan, ear scan, vascular pattern recognition or DNA analysis. The data splitting and/or parser modules of the present invention may be integrated into a wide variety of infrastructure products or applications as desired.

**[0296]** Traditional encryption technologies known in the art rely on one or more key used to encrypt the data and render it unusable without the key. The data, however, remains whole and intact and subject to attack.

The secure data parser of the present invention, in one embodiment, addresses this problem by performing a cryptographic parsing and splitting of the encrypted file into two or more portions or shares, and in another embodiment, preferably four or more shares, adding another layer of encryption to each share of the data, then storing the shares in different physical and/or logical locations. When one or more data shares are physically removed from the system, either by using a removable device, such as a data storage device, or by placing the share under another party's control, any possibility of compromise of secured data is effectively removed.

**[0297]** An example of one embodiment of the secure data parser of the present invention and an example of how it may be utilized is shown in FIGURE 21 and described below. However, it is readily apparent to those of ordinary skill in the art that the secure data parser of the present invention may be utilized in a wide variety of ways in addition to the non-limiting example below. As a deployment option, and in one embodiment, the secure data parser may be implemented with external session key management or secure internal storage of session keys. Upon implementation, a Parser Master Key will be generated which will be used for securing the application and for encryption purposes. It should be also noted that the incorporation of the Parser Master key in the resulting secured data allows for a flexibility of sharing of secured data by individuals within a workgroup, enterprise or extended audience.

[0298] As shown in Figure 21, this embodiment of the present invention shows the steps of the process performed by the secure data parser on data to store the session master key with the parsed data:

[0299] 1. Generating a session master key and encrypt the data using RS1 stream cipher.

[0300] 2. Separating the resulting encrypted data into four shares or portions of parsed data according to the pattern of the session master key.

[0301] 3. In this embodiment of the method, the session master key will be stored along with the secured data shares in a data depository. Separating the session master key according to the pattern of the Parser Master Key and append the key data to the encrypted parsed data.

[0302] 4. The resulting four shares of data will contain encrypted portions of the original data and portions of the session master key. Generate a stream cipher key for each of the four data shares.

[0303] 5. Encrypting each share, then store the encryption keys in different locations from the encrypted data portions or shares: Share 1 gets Key 4, Share 2 gets Key 1, Share 3 gets Key 2, Share 4 gets Key 3.

[0304] To restore the original data format, the steps are reversed.

[0305] It is readily apparent to those of ordinary skill in the art that certain steps of the methods described herein may be performed in different order, or repeated multiple times, as desired. It is also readily apparent to those skilled in the art that the portions of the data may be handled differently from one another.

For example, multiple parsing steps may be performed on only one portion of the parsed data. Each portion of parsed data may be uniquely secured in any desirable way provided only that the data may be reassembled, reconstituted, reformed, decrypted or restored to its original or other usable form.

**[0306]** As shown in FIGURE 22 and described herein, another embodiment of the present invention comprises the steps of the process performed by the secure data parser on data to store the session master key data in one or more separate key management table:

**[0307]** 1. Generating a session master key and encrypt the data using RS1 stream cipher.

**[0308]** 2. Separating the resulting encrypted data into four shares or portions of parsed data according to the pattern of the session master key.

**[0309]** 3. In this embodiment of the method of the present invention, the session master key will be stored in a separate key management table in a data depository. Generating a unique transaction ID for this transaction. Storing the transaction ID and session master key in a separate key management table. Separating the transaction ID according to the pattern of the Parser Master Key and append the data to the encrypted parsed or separated data.

**[0310]** 4. The resulting four shares of data will contain encrypted portions of the original data and portions of the transaction ID.

**[0311]** 5. Generating a stream cipher key for each of the four data shares.

[0312] 6. Encrypting each share, then store the encryption keys in different locations from the encrypted data portions or shares: Share 1 gets Key 4, Share 2 gets Key 1, Share 3 gets Key 2, Share 4 gets Key 3.

[0313] To restore the original data format, the steps are reversed.

[0314] It is readily apparent to those of ordinary skill in the art that certain steps of the method described herein may be performed in different order, or repeated multiple times, as desired. It is also readily apparent to those skilled in the art that the portions of the data may be handled differently from one another. For example, multiple separating or parsing steps may be performed on only one portion of the parsed data. Each portion of parsed data may be uniquely secured in any desirable way provided only that the data may be reassembled, reconstituted, reformed, decrypted or restored to its original or other usable form.

[0315] As shown in Figure 23, this embodiment of the present invention shows the steps of the process performed by the secure data parser on data to store the session master key with the parsed data:

[0316] 1. Accessing the parser master key associated with the authenticated user

[0317] 2. Generating a unique Session Master key

[0318] 3. Derive an Intermediary Key from an exclusive OR function of the Parser Master Key and Session Master key

[0319] 4. Optional encryption of the data using an existing or new encryption algorithm keyed with the Intermediary Key.

[0320] 5. Separating the resulting optionally encrypted data into four shares or portions of parsed data according to the pattern of the Intermediary key.

[0321] 6. In this embodiment of the method, the session master key will be stored along with the secured data shares in a data depository. Separating the session master key according to the pattern of the Parser Master Key and append the key data to the optionally encrypted parsed data shares.

[0322] 7. The resulting multiple shares of data will contain optionally encrypted portions of the original data and portions of the session master key.

[0323] 8. Optionally generate an encryption key for each of the four data shares.

[0324] 9. Optionally encrypting each share with an existing or new encryption algorithm, then store the encryption keys in different locations from the encrypted data portions or shares: for example, Share 1 gets Key 4, Share 2 gets Key 1, Share 3 gets Key 2, Share 4 gets Key 3.

[0325] To restore the original data format, the steps are reversed.

[0326] It is readily apparent to those of ordinary skill in the art that certain steps of the methods described herein may be performed in different order, or repeated multiple times, as desired. It is also readily apparent to those skilled in the art that the portions of

the data may be handled differently from one another. For example, multiple parsing steps may be performed on only one portion of the parsed data. Each portion of parsed data may be uniquely secured in any desirable way provided only that the data may be reassembled, reconstituted, reformed, decrypted or restored to its original or other usable form.

[0327] As shown in FIGURE 24 and described herein, another embodiment of the present invention comprises the steps of the process performed by the secure data parser on data to store the session master key data in one or more separate key management table:

[0328] 1. Accessing the Parser Master Key associated with the authenticated user

[0329] 2. Generating a unique Session Master Key

[0330] 3. Derive an Intermediary Key from an exclusive OR function of the Parser Master Key and Session Master key

[0331] 4. Optionally encrypt the data using an existing or new encryption algorithm keyed with the Intermediary Key.

[0332] 5. Separating the resulting optionally encrypted data into four shares or portions of parsed data according to the pattern of the Intermediary Key.

[0333] 6. In this embodiment of the method of the present invention, the session master key will be stored in a separate key management table in a data depository. Generating a unique transaction ID for this transaction. Storing the transaction ID and session master key in a separate key management table or passing the Session

Master Key and transaction ID back to the calling program for external management. Separating the transaction ID according to the pattern of the Parser Master Key and append the data to the optionally encrypted parsed or separated data.

[0334] 7. The resulting four shares of data will contain optionally encrypted portions of the original data and portions of the transaction ID.

[0335] 8. Optionally generate an encryption key for each of the four data shares.

[0336] 9. Optionally encrypting each share, then store the encryption keys in different locations from the encrypted data portions or shares. For example: Share 1 gets Key 4, Share 2 gets Key 1, Share 3 gets Key 2, Share 4 gets Key 3.

[0337] To restore the original data format, the steps are reversed.

[0338] It is readily apparent to those of ordinary skill in the art that certain steps of the method described herein may be performed in different order, or repeated multiple times, as desired. It is also readily apparent to those skilled in the art that the portions of the data may be handled differently from one another. For example, multiple separating or parsing steps may be performed on only one portion of the parsed data. Each portion of parsed data may be uniquely secured in any desirable way provided only that the data may be reassembled, reconstituted, reformed, decrypted or restored to its original or other usable form.

[0339] A wide variety of encryption methodologies are suitable for use in the methods of the present invention, as is readily apparent to those skilled in the art. The One Time Pad algorithm, is often considered one of the most secure encryption methods, and is suitable for use in the method of the present invention. Using the One Time Pad algorithm requires that a key be generated which is as long as the data to be secured. The use of this method may be less desirable in certain circumstances such as those resulting in the generation and management of very long keys because of the size of the data set to be secured. In the One-Time Pad (OTP) algorithm, the simple exclusive-or function, XOR, is used. For two binary streams x and y of the same length, x XOR y means the bitwise exclusive-or of x and y.

[0340] At the bit level is generated:

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0

[0341] An example of this process is described herein for an n-byte secret, s, (or data set) to be split. The process will generate an n-byte random value, a, and then set:

b = a XOR s.

[0342] Note that one can derive "s" via the equation:

s = a XOR b.

[0343] The values a and b are referred to as shares or portions and are placed in separate depositories. Once

the secret s is split into two or more shares, it is discarded in a secure manner.

**[0344]** The secure data parser of the present invention may utilize this function, performing multiple XOR functions incorporating multiple distinct secret key values: K1, K2, K3, Kn, K5. At the beginning of the operation, the data to be secured is passed through the first encryption operation, secure data = data XOR secret key 5:

S = D XOR K5

**[0345]** In order to securely store the resulting encrypted data in, for example, four shares, S1, S2, S3, Sn, the data is parsed and split into "n" segments, or shares, according to the value of K5. This operation results in "n" pseudorandom shares of the original encrypted data. Subsequent XOR functions may then be performed on each share with the remaining secret key values, for example: Secure data segment 1 = encrypted data share 1 XOR secret key 1:

SD1 = S1 XOR K1

SD2 = S2 XOR K2

SD3 = S3 XOR K3

SDn = Sn XOR Kn.

**[0346]** In one embodiment, it may not be desired to have any one depository contain enough information to decrypt the information held there, so the key required to decrypt the share is stored in a different data depository:

Depository 1: SD1, Kn

Depository 2: SD2, K1

Depository 3: SD3, K2

Depository n: SDn, K3.

**[0347]** Additionally, appended to each share may be the information required to retrieve the original session encryption key, K5. Therefore, in the key management example described herein, the original session master key is referenced by a transaction ID split into "n" shares according to the contents of the installation dependant Parser Master Key (TID1, TID2, TID3, TIDn):

Depository 1: SD1, Kn, TID1

Depository 2: SD2, K1, TID2

Depository 3: SD3, K2, TID3

Depository n: SDn, K3, TIDn.

**[0348]** In the incorporated session key example described herein, the session master key is split into "n" shares according to the contents of the installation dependant Parser Master Key (SK1, SK2, SK3, SKn):

Depository 1: SD1, Kn, SK1

Depository 2: SD2, K1, SK2

Depository 3: SD3, K2, SK3

Depository n: SDn, K3, SKn.

**[0349]** Unless all four shares are retrieved, the data cannot be reassembled according to this example. Even if all four shares are captured, there is no possibility of reassembling or restoring the original information without access to the session master key and the Parser Master Key.

**[0350]** This example has described an embodiment of the method of the present invention, and also describes, in another embodiment, the algorithm used to place shares

into depositories so that shares from all depositories can be combined to form the secret authentication material. The computations needed are very simple and fast. However, with the One Time Pad (OTP) algorithm there may be circumstances that cause it to be less desirable, such as a large data set to be secured, because the key size is the same size as the data to be stored. Therefore, there would be a need to store and transmit about twice the amount of the original data which may be less desirable under certain circumstances.

Stream Cipher RS1

**[0351]** The stream cipher RS1 splitting technique is very similar to the OTP splitting technique described herein. Instead of an n-byte random value, an n′ = min(n, 16)-byte random value is generated and used to key the RS1 Stream Cipher algorithm. The advantage of the RS1 Stream Cipher algorithm is that a pseudorandom key is generated from a much smaller seed number. The speed of execution of the RS1 Stream Cipher encryption is also rated at approximately 10 times the speed of the well known in the art Triple DES encryption without compromising security. The RS1 Stream Cipher algorithm is well known in the art, and may be used to generate the keys used in the XOR function. The RS1 Stream Cipher algorithm is interoperable with other commercially available stream cipher algorithms, such as the RC4™ stream cipher algorithm of RSA Security, Inc and is suitable for use in the methods of the present invention.

**[0352]** Using the key notation above, K1 thru K5 are now an n′ byte random values and we set:

SD1 = S1 XOR E(K1)

SD2 = S2 XOR E(K2)

SD3 = S3 XOR E(K3)

SDn = Sn XOR E(Kn)

where E(K1) thru E(Kn) are the first n′ bytes of output from the RS1 Stream Cipher algorithm keyed by K1 thru Kn. The shares are now placed into data depositories as described herein.

**[0353]** In this stream cipher RS1 algorithm, the required computations needed are nearly as simple and

fast as the OTP algorithm. The benefit in this example using the RS1 Stream Cipher is that the system needs to store and transmit on average only about 16 bytes more than the size of the original data to be secured per share. When the size of the original data is more than 16 bytes, this RS1 algorithm is more efficient than the OTP algorithm because it is simply shorter. It is readily apparent to those of ordinary skill in the art that a wide variety of encryption methods or algorithms are suitable for use in the present invention, including, but not limited to RS1, OTP, RC4™, Triple DES and AES.

**[0354]** There are major advantages provided by the data security methods and computer systems of the present invention over traditional encryption methods. One advantage is the security gained from moving shares of the data to different locations on one or more data depositories or storage devices, that may be in different logical, physical or geographical locations. When the shares of data are split physically and under the control of different personnel, for example, the possibility of compromising the data is greatly reduced.

**[0355]** Another advantage provided by the methods and system of the present invention is the combination of the steps of the method of the present invention for securing data to provide a comprehensive process of maintaining security of sensitive data. The data is encrypted with a secure key and split into one or more shares, and in one embodiment, four shares, according to the secure key. The secure key is stored safely with a reference pointer which is secured into four shares according to a secure

key. The data shares are then encrypted individually and the keys are stored safely with different encrypted shares. When combined, the entire process for securing data according to the methods disclosed herein becomes a comprehensive package for data security.

[0356] The data secured according to the methods of the present invention is readily retrievable and restored, reconstituted, reassembled, decrypted, or otherwise returned into its original or other suitable form for use. In order to restore the original data, the following items may be utilized:

[0357] 1. All shares or portions of the data set.

[0358] 2. Knowledge of and ability to reproduce the process flow of the method used to secure the data.

[0359] 3. Access to the session master key.

[0360] 4. Access to the Parser Master Key.

[0361] Therefore, it may be desirable to plan a secure installation wherein at least one of the above elements may be physically separated from the remaining components of the system (under the control of a different system administrator for example).

[0362] Protection against a rogue application invoking the data securing methods application may be enforced by use of the Parser Master Key. A mutual authentication handshake between the secure data parser and the application may be required in this embodiment of the present invention prior to any action taken.

[0363] The security of the system dictates that there be no "backdoor" method for recreation of the original data. For installations where data recovery issues may

arise, the secure data parser can be enhanced to provide a mirror of the four shares and session master key depository. Hardware options such as RAID (redundant array of inexpensive disks, used to spread information over several disks) and software options such as replication can assist as well in the data recovery planning.

Key Management

**[0364]** In one embodiment of the present invention, the data securing method uses three sets of keys for an encryption operation. Each set of keys may have individual key storage, retrieval, security and recovery options, based on the installation. The keys that may be used, include, but are not limited to:

The Parser Master Key

**[0365]** This key is an individual key associated with the installation of the secure data parser. It is installed on the server on which the secure data parser has been deployed. There are a variety of options suitable for securing this key including, but not limited to, a smart card, separate hardware key store, standard key stores, custom key stores or within a secured database table, for example.

The Session Master Key

**[0366]** A Session Master Key may be generated each time data is secured. The Session Master Key is used to encrypt the data prior to the parsing and splitting operations. It may also be incorporated (if the Session Master Key is not integrated into the parsed data) as a

means of parsing the encrypted data. The Session Master Key may be secured in a variety of manners, including, but not limited to, a standard key store, custom key store, separate database table, or secured within the encrypted shares, for example.

The Share Encryption Keys

[0367] For each share or portions of a data set that is created, an individual Share Encryption Key may be generated to further encrypt the shares. The Share Encryption Keys may be stored in different shares than the share that was encrypted.

[0368] It is readily apparent to those of ordinary skill in the art that the data securing methods and computer system of the present invention are widely applicable to any type of data in any setting or environment. In addition to commercial applications conducted over the Internet or between customers and vendors, the data securing methods and computer systems of the present invention are highly applicable to non-commercial or private settings or environments. Any data set that is desired to be kept secure from any unauthorized user may be secured using the methods and systems described herein. For example, access to a particular database within a company or organization may be advantageously restricted to only selected users by employing the methods and systems of the present invention for securing data. Another example is the generation, modification or access to documents wherein it is desired to restrict access or prevent unauthorized or accidental access or disclosure outside a group of

selected individuals, computers or workstations. These and other examples of the ways in which the methods and systems of data securing of the present invention are applicable to any non-commercial or commercial environment or setting for any setting, including, but not limited to any organization, government agency or corporation.

**[0369]** In another embodiment of the present invention, the data securing method uses three sets of keys for an encryption operation. Each set of keys may have individual key storage, retrieval, security and recovery options, based on the installation. The keys that may be used, include, but are not limited to:

1. The Parser Master Key

**[0370]** This key is an individual key associated with the installation of the secure data parser. It is installed on the server on which the secure data parser has been deployed. There are a variety of options suitable for securing this key including, but not limited to, a smart card, separate hardware key store, standard key stores, custom key stores or within a secured database table, for example.

2. The Session Master Key

**[0371]** A Session Master Key may be generated each time data is secured. The Session Master Key is used in conjunction with the Parser Master key to derive the Intermediary Key. The Session Master Key may be secured in a variety of manners, including, but not limited to, a standard key store, custom key store, separate database

table, or secured within the encrypted shares, for example.

3. The Intermediary Key

**[0372]** An Intermediary Key may be generated each time data is secured. The Intermediary Key is used to encrypt the data prior to the parsing and splitting operation. It may also be incorporated as a means of parsing the encrypted data.

4. The Share Encryption Keys

**[0373]** For each share or portions of a data set that is created, an individual Share Encryption Key may be generated to further encrypt the shares. The Share Encryption Keys may be stored in different shares than the share that was encrypted.

**[0374]** It is readily apparent to those of ordinary skill in the art that the data securing methods and computer system of the present invention are widely applicable to any type of data in any setting or environment. In addition to commercial applications conducted over the Internet or between customers and vendors, the data securing methods and computer systems of the present invention are highly applicable to non-commercial or private settings or environments. Any data set that is desired to be kept secure from any unauthorized user may be secured using the methods and systems described herein. For example, access to a particular database within a company or organization may be advantageously restricted to only selected users by employing the methods and systems of the present invention for securing data. Another example is the

generation, modification or access to documents wherein it is desired to restrict access or prevent unauthorized or accidental access or disclosure outside a group of selected individuals, computers or workstations. These and other examples of the ways in which the methods and systems of data securing of the present invention are applicable to any non-commercial or commercial environment or setting for any setting, including, but not limited to any organization, government agency or corporation.

Workgroup, Project, Individual PC/Laptop or Cross Platform Data Security

**[0375]** The data securing methods and computer systems of the present invention are also useful in securing data by workgroup, project, individual PC/Laptop and any other platform that is in use in, for example, businesses, offices, government agencies, or any setting in which sensitive data is created, handled or stored. The present invention provides methods and computer systems to secure data that is known to be sought after by organizations, such as the U.S. Government, for implementation across the entire government organization or between governments at a state or federal level.

**[0376]** The data securing methods and computer systems of the present invention provide the ability to not only parse and split flat files but also data fields, sets and or table of any type. Additionally, all forms of data are capable of being secured under this process, including, but not limited to, text, video, images, biometrics and voice data. Scalability, speed and data throughput of

the methods of securing data of the present invention are only limited to the hardware the user has at their disposal.

[0377] In one embodiment of the present invention, the data securing methods are utilized as described below in a workgroup environment. In one embodiment, as shown in FIGURE 23 and described below, the Workgroup Scale data securing method of the present invention uses the private key management functionality of the TrustEngine to store the user/group relationships and the associated private keys (Parser Group Master Keys) necessary for a group of users to share secure data. The method of the present invention has the capability to secure data for an enterprise, workgroup, or individual user, depending on how the Parser Master Key was deployed.

[0378] In one embodiment, additional key management and user/group management programs may be provided, enabling wide scale workgroup implementation with a single point of administration and key management. Key generation, management and revocation are handled by the single maintenance program, which all become especially important as the number of users increase. In another embodiment, key management may also be set up across one or several different system administrators, which may not allow any one person or group to control data as needed. This allows for the management of secured data to be obtained by roles, responsibilities, membership, rights, etc., as defined by an organization, and the access to secured data can be limited to just those who are permitted or required to have access only to the portion

they are working on, while others, such as managers or executives, may have access to all of the secured data. This embodiment allows for the sharing of secured data among different groups within a company or organization while at the same time only allowing certain selected individuals, such as those with the authorized and predetermined roles and responsibilities, to observe the data as a whole. In addition, this embodiment of the methods and systems of the present invention also allows for the sharing of data among, for example, separate companies, or separate departments or divisions of companies, or any separate organization departments, groups, agencies, or offices, or the like, of any government or organization or any kind, where some sharing is required, but not any one party may be permitted to have access to all the data. Particularly apparent examples of the need and utility for such a method and system of the present invention are to allow sharing, but maintain security, in between government areas, agencies and offices, and between different divisions, departments or offices of a large company, or any other organization, for example.

**[0379]** An example of the applicability of the methods of the present invention on a smaller scale is as follows. A Parser Master key is used as a serialization or branding of the secure data parser to an organization. As the scale of use of the Parser Master key is reduced from the whole enterprise to a smaller workgroup, the data securing methods described herein are used to share files within groups of users.

[0380] In the example shown in FIGURE 25 and described below, there are six users defined along with their title or role within the organization. The side bar represents five possible groups that the users can belong to according to their role. The arrow represents membership by the user in one or more of the groups.

[0381] When configuring the secure data parser for use in this example, the system administrator accesses the user and group information from the operating system by a maintenance program. This maintenance program generates and assigns Parser Group Master Keys to users based on their membership in groups.

[0382] In this example, there are three members in the Senior Staff group. For this group, the actions would be:

[0383] 1. Access Parser Group Master Key for the Senior Staff group (generate a key if not available);

[0384] 2. Generate a digital certificate associating CEO with the Senior Staff group;

[0385] 3. Generate a digital certificate associating CFO with the Senior Staff group;

[0386] 4. Generate a digital certificate associating Vice President, Marketing with the Senior Staff group.

[0387] The same set of actions would be done for each group, and each member within each group. When the maintenance program is complete, the Parser Group Master Key becomes a shared credential for each member of the group. Revocation of the assigned digital certificate may be done automatically when a user is removed from a

group through the maintenance program without affecting the remaining members of the group.

[0388] Once the shared credentials have been defined, the parsing and splitting process remains the same. When a file, document or data element is to be secured, the user is prompted for the target group to be used when securing the data. The resulting secured data is only accessible by other members of the target group. This functionality of the methods and systems of the present invention may be used with any other computer system or software platform, any may be, for example, integrated into existing application programs or used standalone for file security.

[0389] It is readily apparent to those of ordinary skill in the art that any one or combination of encryption algorithms are suitable for use in the methods and systems of the present invention. For example, the encryption steps may, in one embodiment, be repeated to produce a multi-layered encryption scheme. In addition, a different encryption algorithm, or combination of encryption algorithms, may be used in repeat encryption steps such that different encryption algorithms are applied to the different layers of the multi-layered encryption scheme. As such, the encryption scheme itself may become a component of the methods of the present invention for securing sensitive data from unauthorized use or access.

[0390] The secure data parser may include as an internal component, as an external component, or as both an error-checking component. For example, in one

suitable approach, as portions of data are created using the secure data parser in accordance with the present invention, to assure the integrity of the data within a portion, a hash value is taken at preset intervals within the portion and is appended to the end of the interval. The hash value is a predictable and reproducible numeric representation of the data. If any bit within the data changes, the hash value would be different. A scanning module (either as a stand-alone component external to the secure data parser or as an internal component) may then scan the portions of data generated by the secure data parser. Each portion of data (or alternatively, less than all portions of data according to some interval or by a random or pseudo-random sampling) is compared to the appended hash value or values and an action may be taken. This action may include a report of values that match and do not match, an alert for values that do not match, or invoking of some external or internal program to trigger a recovery of the data. For example, recovery of the data could be performed by invoking a recovery module based on the concept that fewer than all portions may be needed to generate original data in accordance with the present invention.

**[0391]** The secure data parser of the present invention may be used in any suitable application. Namely, the secure data parser described herein has a variety of applications in different areas of computing and technology. Several such areas are discussed below. It will be understood that these are merely illustrative in nature and that any other suitable applications may make

use of the secure data parser. It will further be understood that the examples described are merely illustrative embodiments that may be modified in any suitable way in order to satisfy any suitable desires. For example, parsing and splitting may be based on any suitable units, such as by bits, by bytes, by kilobytes, by megabytes, by any combination thereof, or by any other suitable unit.

**[0392]** The secure data parser of the present invention may be used to implement secure physical tokens, whereby data stored in a physical token may be required in order to access additional data stored in another storage area. In one suitable approach, a physical token, such as a compact USB flash drive, a floppy disk, an optical disk, a smart card, or any other suitable physical token, may be used to store one of at least two portions of parsed data in accordance with the present invention. In order to access the original data, the USB flash drive would need to be accessed. Thus, a personal computer holding one portion of parsed data would need to have the USB flash drive, having the other portion of parsed data, attached before the original data can be accessed. FIGURE 26 illustrates this application. Storage area 2500 includes a portion of parsed data 2502. Physical token 2504, having a portion of parsed data 2506 would need to be coupled to storage area 2500 using any suitable communications interface 2508 (e.g., USB, serial, parallel, bluetooth, IR, IEEE 1394, ethernet, or any other suitable communications interface) in order to access the original data. This is useful in a situation

where, for example, sensitive data on a computer is left alone and subject to unauthorized access attempts. By removing the physical token (e.g., the USB flash drive), the sensitive data is inaccessible. It will be understood that any other suitable approach for using physical tokens may be used.

[0393] The secure data parser of the present invention may be used to implement a secure authentication system whereby user enrollment data (e.g., passwords, private encryption keys, fingerprint templates, biometric data or any other suitable user enrollment data) is parsed and split using the secure data parser. The user enrollment data may be parsed and split whereby one or more portions are stored on a smart card, a government Common Access Card, any suitable physical storage device (e.g., magnetic or optical disk, USB key drive, etc.), or any other suitable device. One or more other portions of the parsed user enrollment data may be stored in the system performing the authentication. This provides an added level of security to the authentication process (e.g., in addition to the biometric authentication information obtained from the biometric source, the user enrollment data must also be obtained via the appropriate parsed and split data portion).

[0394] The secure data parser of the present invention may be integrated into any suitable existing system in order to provide the use of its functionality in each system's respective environment. FIGURE 27 shows a block diagram of an illustrative system 2600, which may include software, hardware, or both for implementing any suitable

application. System 2600 may be an existing system in which secure data parser 2602 may be retrofitted as an integrated component. Alternatively, secure data parser 2602 may be integrated into any suitable system 2600 from, for example, its earliest design stage. Secure data parser 2600 may be integrated at any suitable level of system 2600. For example, secure data parser 2602 may be integrated into system 2600 at a sufficiently back-end level such that the presence of secure data parser 2602 may be substantially transparent to an end user of system 2600. Secure data parser 2602 may be used for parsing and splitting data among one or more storage devices 2604 in accordance with the present invention. Some illustrative examples of systems having the secure data parser integrated therein are discussed below.

**[0395]** The secure data parser of the present invention may be integrated into an operating system kernel (e.g., Linux, Unix, or any other suitable commercial or proprietary operating system). This integration may be used to protect data at the device level whereby, for example, data that would ordinarily be stored in one or more devices is separated into a certain number of portions by the secure data parser integrated into the operating system and stored among the one or more devices. When original data is attempted to be accessed, the appropriate software, also integrated into the operating system, may recombine the parsed data portions into the original data in a way that may be transparent to the end user.

[0396] The secure data parser of the present invention may be integrated into a volume manager or any other suitable component of a storage system to protect local and networked data storage across any or all supported platforms. For example, with the secure data parser integrated, a storage system may make use of the redundancy offered by the secure data parser (i.e., which is used to implement the feature of needing fewer than all separated portions of data in order to reconstruct the original data) to protect against data loss. The secure data parser also allows all data written to storage devices, whether using redundancy or not, to be in the form of multiple portions that are generated according to the parsing of the present invention. When original data is attempted to be accessed, the appropriate software, also integrated into the volume manager or other suitable component of the storage system, may recombine the parsed data portions into the original data in a way that may be transparent to the end user.

[0397] In one suitable approach, the secure data parser of the present invention may be integrated into a RAID controller (as either hardware or software). This allows for the secure storage of data to multiple drives while maintaining fault tolerance in case of drive failure.

[0398] The secure data parser of the present invention may be integrated into a database in order to, for example, protect sensitive table information. For example, in one suitable approach, data associated with

particular cells of a database table (e.g., individual cells, one or more particular columns, one or more particular rows, any combination thereof, or an entire database table) may be parsed and separated according to the present invention (e.g., where the different portions are stored on one or more storage devices at one or more locations or on a single storage device). Access to recombine the portions in order to view the original data may be granted by traditional authentication methods (e.g., username and password query).

**[0399]** The secure parser of the present invention may be integrated in any suitable system that involves data in motion (i.e., transfer of data from one location to another). Such systems include, for example, email, streaming data broadcasts, and wireless (e.g., WiFi) communications. With respect to email, in one suitable approach, the secure parser may be used to parse outgoing messages (i.e., containing text, binary data, or both (e.g., files attached to an email message)) and sending the different portions of the parsed data along different paths thus creating multiple streams of data. If any one of these streams of data is compromised, the original message remains secure because the system may require that more than one of the portions be combined, in accordance with the present invention, in order to generate the original data. In another suitable approach, the different portions of data may be communicated along one path sequentially so that if one portion is obtained, it may not be sufficient to generate the original data. The different portions arrive at the

intended recipient's location and may be combined to generate the original data in accordance with the present invention.

**[0400]** FIGURES 27 and 28 are illustrative block diagrams of such email systems. FIGURE 28 shows a sender system 2700, which may include any suitable hardware, such as a computer terminal, personal computer, handheld device (e.g., PDA), cellular telephone, computer network, any other suitable hardware, or any combination thereof. Sender system 2700 is used to generate and/or store a message 2704, which may be, for example, an email message, a binary data file (e.g., graphics, voice, video, etc.), or both. Message 2704 is parsed and split by secure data parser 2702 in accordance with the present invention. The resultant data portions may be communicated across one or more separate communications paths 2706 over network 2708 (e.g., the Internet, an intranet, a LAN, WiFi, bluetooth, any other suitable hard-wired or wireless communications means, or any combination thereof) to recipient system 2710. The data portions may be communicated parallel in time or alternatively, according to any suitable time delay between the communication of the different data portions. Recipient system 2710 may be any suitable hardware as described above with respect to sender system 2700. The separate data portions carried along communications paths 2706 are recombined at recipient system 2710 to generate the original message or data in accordance with the present invention.

**[0401]** FIGURES 28 shows a sender system 2800, which may include any suitable hardware, such as a computer terminal, personal computer, handheld device (e.g., PDA), cellular telephone, computer network, any other suitable hardware, or any combination thereof. Sender system 2800 is used to generate and/or store a message 2804, which may be, for example, an email message, a binary data file (e.g., graphics, voice, video, etc.), or both. Message 2804 is parsed and split by secure data parser 2802 in accordance with the present invention. The resultant data portions may be communicated across a single communications paths 2806 over network 2808 (e.g., the Internet, an intranet, a LAN, WiFi, bluetooth, any other suitable communications means, or any combination thereof) to recipient system 2810. The data portions may be communicated serially across communications path 2806 with respect to one another. Recipient system 2810 may be any suitable hardware as described above with respect to sender system 2800. The separate data portions carried along communications path 2806 are recombined at recipient system 2810 to generate the original message or data in accordance with the present invention.

**[0402]** It will be understood that the arrangement of FIGURES 27 and 28 are merely illustrative. Any other suitable arrangement may be used. For example, in another suitable approach, the systems of FIGURES 27 and 28 may be incorporated whereby the multi-path approach of FIGURE 28 is used and in which one or more of communications paths 2706 are used to carry more than one

portion of data as communications path 2806 does in the context of FIGURE 29.

**[0403]** The secure data parser may be integrated at any suitable level of a data-in motion system. For example, in the context of an email system, the secure parser may be integrated at the user-interface level (e.g., into Microsoft® Outlook) in which case, the user may have control over the use of the secure data parser features when using email. Alternatively, the secure parser may be implemented in a back-end component such as at the exchange server, in which case, messages may be automatically parsed, split, and communicated along different paths in accordance with the present invention without any user intervention.

**[0404]** Similarly, in the case of streaming broadcasts of data (e.g., audio, video), the outgoing data may be parsed and separated into multiple streams each containing a portion of the parsed data. The multiple streams may be transmitted along one or more paths and recombined at the recipient's location in accordance with the present invention. One of the benefits of this approach is that it avoids the relatively large overhead associated with traditional encryption of data followed by transmission of the encrypted data over a single communications channel. The secure parser of the present invention allows data in motion to be sent in multiple parallel streams, increasing speed and efficiency.

**[0405]** The secure data parser of the present invention may provide security with the ability for disaster recovery. According to the present invention, fewer than

all portions of the separated data generated by the secure data parser may be necessary in order to retrieve the original data. That is, out of m portions stored, n may be the minimum number of these m portions necessary to retrieve the original data, where n <= m. For example, if each of four portions is stored in a different physical location relative to the other three portions, then, if n=2 in this example, two of the locations may be compromised whereby data is destroyed or inaccessible, and the original data may still be retrieved from the portions in the other two locations. Any suitable value for n or m may be used.

**[0406]** In addition, the n of m feature of the present invention may be used to create a "two man rule" whereby in order to avoid entrusting a single individual or any other entity with full access to what may be sensitive data, two or more distinct entities, each with a portion of the separated data parsed by the secure parser of the present invention may need to agree to put their portions together in order to retrieve the original data.

**[0407]** The secure data parser of the present invention may be used to provide a group of entities with a group-wide key that allows the group members to access particular information authorized to be accessed by that particular group. The group key may be one of the data portions generated by the secure parser in accordance with the present invention that may be required to be combined with another portion centrally stored, for example in order to retrieve the information sought. This feature allows for, for example, secure

collaboration among a group. It may be applied in for example, dedicated networks, virtual private networks, intranets, or any other suitable network.

**[0408]** Specific applications of this use of the secure parser include, for example, coalition information sharing in which, for example, multi-national friendly government forces are given the capability to communicate operational and otherwise sensitive data on a security level authorized to each respective country over a single network or a dual network (i.e., as compared to the many networks involving relatively substantial manual processes currently used). This capability is also applicable for companies or other organizations in which information needed to be known by one or more specific individuals (within the organization or without) may be communicated over a single network without the need to worry about unauthorized individuals viewing the information.

**[0409]** Another specific application includes a multi-level security hierarchy for government systems. That is, the secure parser of the present invention may provide for the ability to operate a government system at different levels of classified information (e.g., unclassified, classified, secret, top secret) using a single network. If desired, more networks may be used (e.g., a separate network for top secret), but the present invention allows for substantially fewer than current arrangement in which a separate network is used for each level of classification.

**[0410]** It will be understood that any combination of the above described applications of the secure parser of the present invention may be used. For example, the group key application can be used together with the data in motion security application (i.e., whereby data that is communicated over a network can only be accessed by a member of the respective group and where, while the data is in motion, it is split among multiple paths (or sent in sequential portions) in accordance with the present invention).

**[0411]** The secure data parser of the present invention may be integrated into any middleware application to enable applications to securely store data to different database products or to different devices without modification to either the applications or the database. Middleware is a general term for any product that allows two separate and already existing programs to communicate. For example, in one suitable approach, middleware having the secure data parser integrated, may be used to allow programs written for a particular database to communicate with other databases without custom coding.

**[0412]** The secure data parser of the present invention may be implemented having any combination of any suitable capabilities, such as those discussed herein. In some embodiments of the present invention, for example, the secure data parser may be implemented having only certain capabilities whereas other capabilities may be obtained through the use of external software, hardware, or both

interfaced either directly or indirectly with the secure data parser.

**[0413]** FIGURE 30, for example, shows an illustrative implementation of the secure data parser as secure data parser 3000. Secure data parser 3000 is implemented with very few built-in capabilities. As illustrated, secure data parser 3000 may include built-in capabilities for parsing and splitting data into portions (also referred to herein as shares) of data using module 3002 in accordance with the present invention. Secure data parser 3000 may also include built in capabilities for performing redundancy in order to be able to implement, for example, the m of n feature described above (i.e., recreating the original data using fewer than all shares of parsed and split data) using module 3004. Secure data parser 3000 may also include share distribution capabilities using module 3006 for placing the shares of data into buffers from which they are sent for communication to a remote location, for storage, etc. in accordance with the present invention. It will be understood that any other suitable capabilities may be built into secure data parser 3000.

**[0414]** Assembled data buffer 3008 may be any suitable memory used to store the original data (although not necessarily in its original form) that will be parsed and split by secure data parser 3000. In a splitting operation, assembled data buffer 3008 provides input to secure data parser 3008. In a restore operation, assembled data buffer 3008 may be used to store the output of secure data parser 3000.

[0415] Split shares buffers 3010 may be one or more memory modules that may be used to store the multiple shares of data that resulted from the parsing and splitting of original data. In a splitting operation, split shares buffers 3010 hold the output of the secure data parser. In a restore operation, split shares buffers hold the input to secure data parser 3000.

[0416] It will be understood that any other suitable arrangement of capabilities may be built-in for secure data parser 3000. Any additional features may be built-in and any of the features illustrated may be removed, made more robust, made less robust, or may otherwise be modified in any suitable way. Buffers 3008 and 3010 are likewise merely illustrative and may be modified, removed, or added to in any suitable way.

[0417] Any suitable modules implemented in software, hardware or both may be called by or may call to secure data parser 3000. If desired, even capabilities that are built into secure data parser 3000 may be replaced by one or more external modules. As illustrated, some external modules include random number generator 3012, cipher feedback key generator 3014, hash algorithm 3016, any one or more types of encryption 3018, and key management 3020. It will be understood that these are merely illustrative external modules. Any other suitable modules may be used in addition to or in place of those illustrated.

[0418] Cipher feedback key generator 3014 may, externally to secure data parser 3000, generate for each secure data parser operation, a unique key, or random

number (using for example random number generator 3012), to be used as a seed value for an operation that extends an original session key size (e.g., a value of 128, 256, 512, or 1024 bits) into a value equal to the length of the data to be parsed and split. Any suitable algorithm may be used for the cipher feedback key generation, including, for example, the AES cipher feedback key generation algorithm.

**[0419]** In order to facilitate integration of secure data parser 3000 and its external modules (i.e., secure data parser layer 3026) into an application layer 3024 (e.g., email application, database application, etc.), a wrapping layer that may make use of, for example, API function calls may be used. Any other suitable arrangement for facilitating integration of secure data parser layer 3026 into application layer 3024 may be used.

**[0420]** FIGURE 31 illustratively shows how the arrangement of FIGURE 30 may be used when a write (e.g., to a storage device), insert (e.g., in a database field), or transmit (e.g., across a network) command is issued in application layer 3024. At step 3100 data to be secured is identified and a call is made to the secure data parser. The call is passed through wrapper layer 3022 where at step 3102, wrapper layer 3022 streams the input data identified at step 3100 into assembled data buffer 3008. Also at step 3102, any suitable share information, filenames, any other suitable information, or any combination thereof may be stored (e.g., as information 3106 at wrapper layer 3022). Secure data processor 3000

then parses and splits the data it takes as input from assembled data buffer 3008 in accordance with the present invention. It outputs the data shares into split shares buffers 3010. At step 3104, wrapper layer 3022 obtains from stored information 3106 any suitable share information (i.e., stored by wrapper 3022 at step 3102) and share location(s) (e.g., from one or more configuration files). Wrapper layer 3022 then writes the output shares (obtained from split shares buffers 3010) appropriately (e.g., written to one or more storage devices, communicated onto a network, etc.).

**[0421]** FIGURE 32 illustratively shows how the arrangement of FIGURE 30 may be used when a read (e.g., from a storage device), select (e.g., from a database field), or receive (e.g., from a network) occurs. At step 3200, data to be restored is identified and a call to secure data parser 3000 is made from application layer 3024. At step 3202, from wrapper layer 3022, any suitable share information is obtained and share location is determined. Wrapper layer 3022 loads the portions of data identified at step 3200 into split shares buffers 3010. Secure data parser 3000 then processes these shares in accordance with the present invention (e.g., if only three of four shares are available, then the redundancy capabilities of secure data parser 3000 may be used to restore the original data using only the three shares). The restored data is then stored in assembled data buffer 3008. At step 3204, application layer 3022 converts the data stored in assembled data buffer 3008 into its original data format (if necessary) and provides

the original data in its original format to application layer 3024.

**[0422]** It will be understood that the parsing and splitting of original data illustrated in FIGURE 31 and the restoring of portions of data into original data illustrated in FIGURE 32 is merely illustrative. Any other suitable processes, components, or both may be used in addition to or in place of those illustrated.

**[0423]** FIGURE 33 is a block diagram of an illustrative process flow for parsing and splitting original data into two or more portions of data in accordance with one embodiment of the present invention. As illustrated, the original data desired to be parsed and split is plain text 3306 (i.e., the word "SUMMIT" is used as an example). It will be understood that any other type of data may be parsed and split in accordance with the present invention. A session key 3300 is generated. If the length of session key 3300 is not compatible with the length of original data 3306, then cipher feedback session key 3304 may be generated.

**[0424]** In one suitable approach, original data 3306 may be encrypted prior to parsing, splitting, or both. For example, as FIGURE 33 illustrates, original data 3306 may be XORed with any suitable value (e.g., with cipher feedback session key 3304, or with any other suitable value). It will be understood that any other suitable encryption technique may be used in place of or in addition to the XOR technique illustrate. It will further be understood that although FIGURE 33 is illustrated in terms of byte by byte operations, the

operation may take place at the bit level or at any other suitable level. It will further be understood that, if desired, there need not be any encryption whatsoever of original data 3306.

**[0425]** The resultant encrypted data (or original data if no encryption took place) is then hashed to determine how to split the encrypted (or original) among the output buckets (e.g., of which there are four in the illustrated example). In the illustrated example, the hashing takes place by bytes and is a function of cipher feedback session key 3304. It will be understood that this is merely illustrative. The hashing may be performed at the bit level, if desired. The hashing may be a function of any other suitable value besides cipher feedback session key 3304. In another suitable approach, hashing need not be used. Rather, any other suitable technique for splitting data may be employed.

**[0426]** FIGURE 34 is a block diagram of an illustrative process flow for restoring original data 3306 from two or more parsed and split portions of original data 3306 in accordance with one embodiment of the present invention. The process involves hashing the portions in reverse (i.e., to the process of FIGURE 33) as a function of cipher feedback session key 3304 to restore the encrypted original data (or original data if there was no encryption prior to the parsing and splitting). The encryption key may then be used to restore the original data (i.e., in the illustrated example, cipher feedback session key 3304 is used to decrypt the XOR encryption by

XORing it with the encrypted data). This the restores original data 3306.

[0427] FIGURE 35 shows how bit-splitting may be implemented in the example of FIGURES 33 and 34. A hash may be used (e.g., as a function of the cipher feedback session key, as a function of any other suitable value) to determine a bit value at which to split each byte of data. It will be understood that this is merely one illustrative way in which to implement splitting at the bit level. Any other suitable technique may be used.

[0428] It will be understood that any reference to hash functionality made herein may be made with respect to any suitable hash algorithm. These include for example, MD5 and SHA-1. Different hash algorithms may be used at different times and by different components of the present invention.

[0429] In one embodiment of the present invention, an audit module may be provided that tracks whenever the secure data parser is invoked to parse and/or split data.

[0430] Additionally, other combinations, additions, substitutions and modifications will be apparent to the skilled artisan in view of the disclosure herein. Accordingly, the present invention is not intended to be limited by the reaction of the preferred embodiments but is to be defined by a reference to the appended claims.

[0431] It shall be understood that features, approaches, embodiments, arrangements, etc. as described in this provisional application shall take on illustrative interpretations. For example, when language

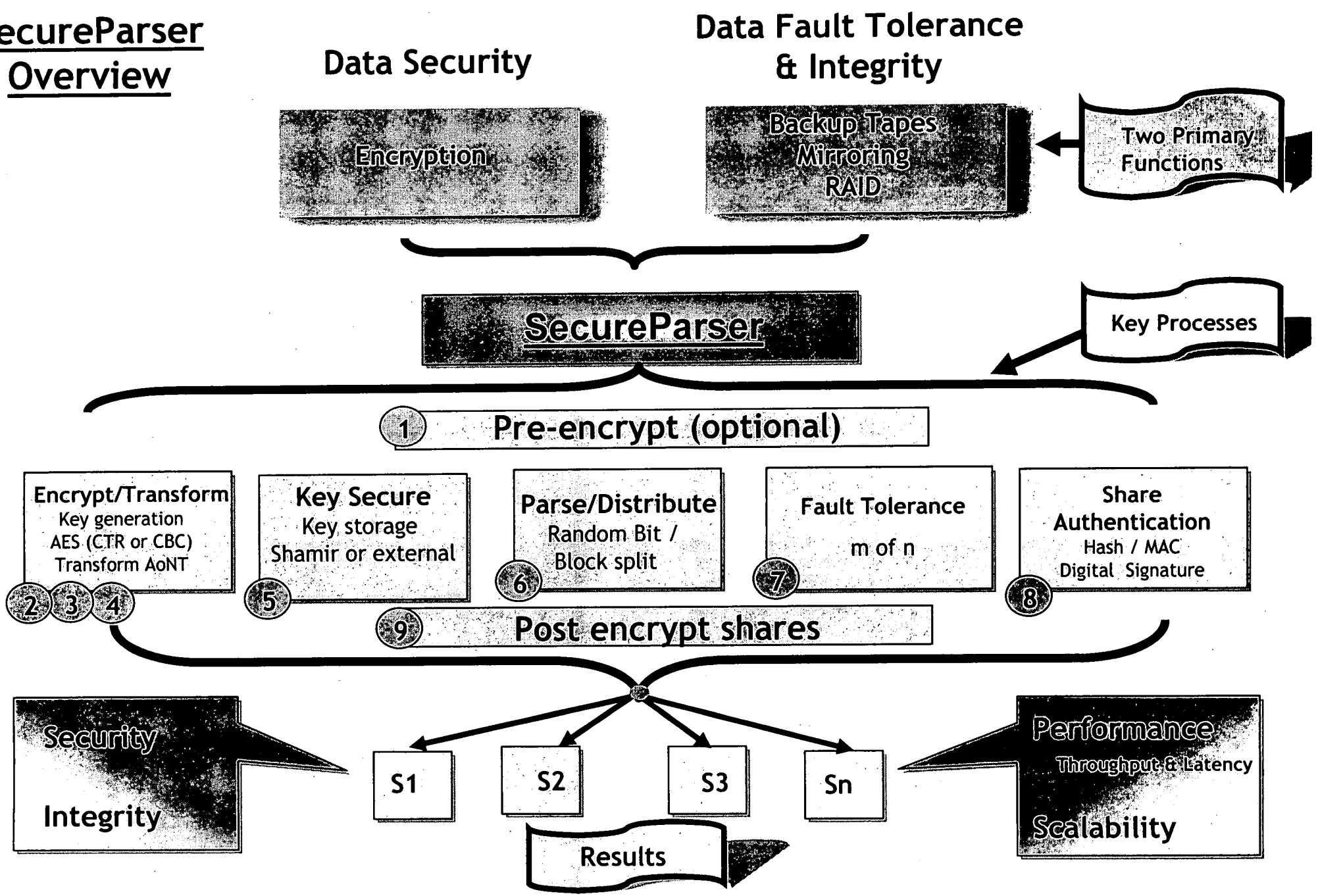
such as "required," "must," and "is" is used, it shall be interpreted as meaning "may be."

# SECURE DATA PARSER METHOD AND SYSTEM

Abstract of the Invention

[0432] A secure data parser is provided that may be integrated into any suitable system for securely storing and communicating data. The secure data parser parses data and then splits the data into multiple portions that are stored or communicated distinctly. Encryption of the original data, the portions of data, or both may be employed for additional security. The secure data parser may be used to protect data in motion by splitting original data into portions of data, that may be communicated using multiple communications paths.

# SecureParser Overview



1. **Pre-encryption (optional):** The plain text is optionally encrypted using AES under a 128, 192 or 256-bit key. This supports FIPS Compliance and compatibility with sites using external key management systems.

2. **Encryption Split Session Key Generation:** A session key is generated using a cryptographically secure pseudo-random number generator algorithm (CSPRNG). This key is unique for each session and may be 128, 192 or 256 bits in length.

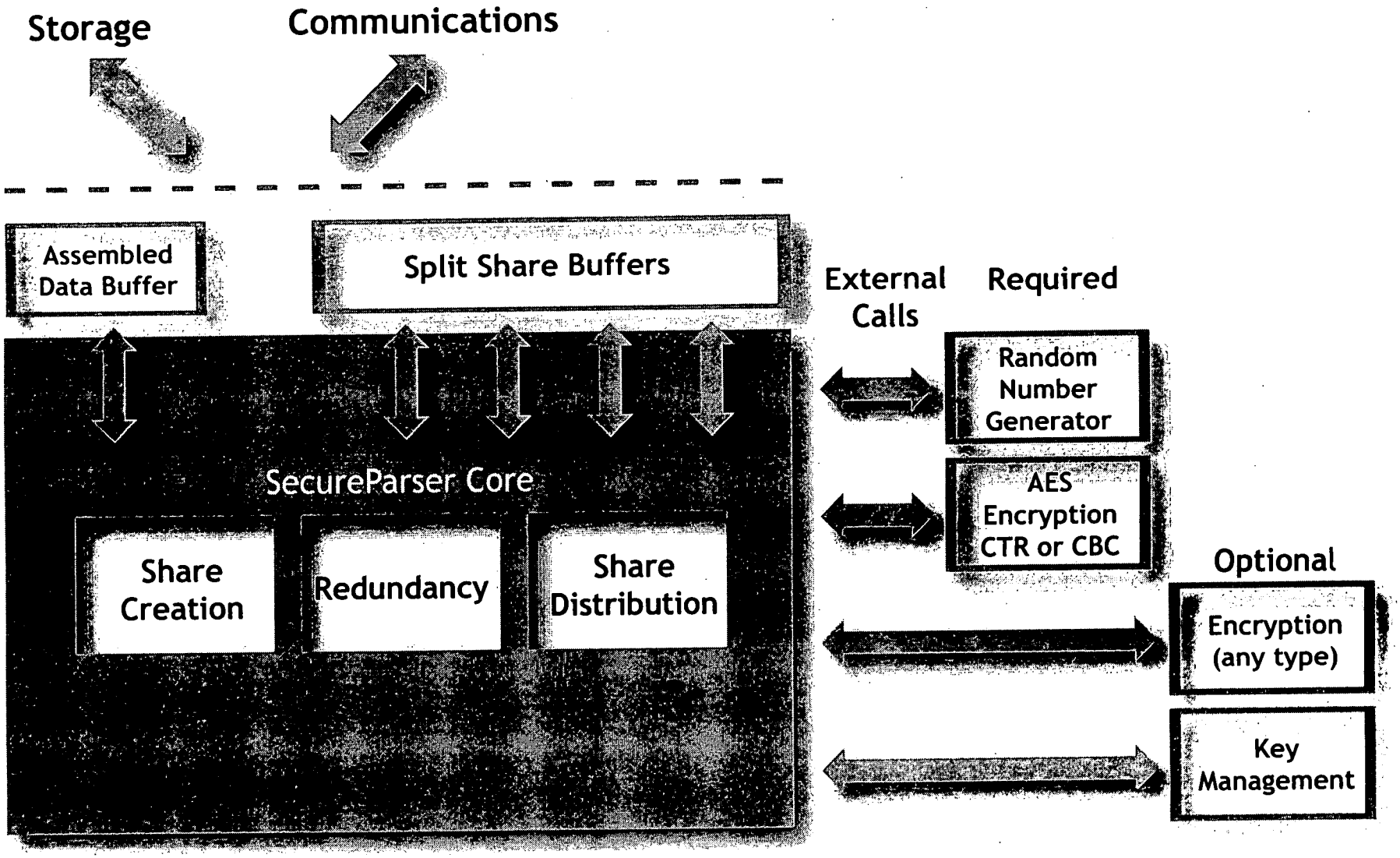
3. **Optionally Pre-encrypted:** data is encrypted in its entirety using the AES encryption algorithm and under the split session key. Other encryption algorithms can be integrated to support government or commercial encryption requirements.

4. **Transform:** The split session encryption key is processed using the Package Transform, or a form of "All or Nothing Transform (AoNT)". This procedure prevents key exposure when fewer than the required shares are exposed.

5. **Key Secure:** The above transform key or split session key is securely divided into key-shares using a "split knowledge" key splitting technique such as the Shamir algorithm. Each key-share is distributed to one data share.
6. **Parse / Distribute:** The data is partitioned into segments using one of several distribution algorithms. The data may be distributed to shares randomly at the bit or block level.

7. **Fault Tolerance:** When fault tolerant mode is used, additional information is written to each share to allow for restoration of the data from less than the total number of shares. This method is referred to as "m of n"
8. **Integrity:** Integrity information is written to each share in the form of a hash value, which allows for the detection of damaged or malformed shares. For additional protection, the shares may also be protected by use of a digital signature or Message Authentication Code (MAC).

9. **Post-encryption (optional):** The resulting shares may optionally be encrypted under a key provided by the user.


Storage
Communications
Assembled Data Buffer
Split Share Buffers
SecureParser Core
Share Creation
Redundancy
Share Distribution
External Calls
Required
Random Number Generator
AES Encryption CTR or CBC
Optional
Encryption (any type)
Key Management

# SecureParser Toolkit Combination Options

- **The SecureParser Toolkit has over 1,200 combinations of options to choose from when determining what type of data Transformation, Share Distribution scheme, Fault Tolerance Mode, Key Secure and Share Authentication is desired.**
  - 123 combinations using the AoNT transform & AES CTR Mode
  - 123 combinations using DES Encryption
  - 123 combinations using 3Des Encryption
  - 123 combinations using AES counter mode
  - 123 combinations for each other encryption one chooses
  - 369 combinations if choose to not Transform data
    - Not shown in these charts but available.
  - 246 combinations based on various key lengths
  - Hundreds of other combinations are available because of ability to link with just about all other encryptions type.
    - Additional Parsring techniques are under development which will be a toolkit combination options
- All options are recommended to be done based on the threat and Risk Scenario attempting to secure data from
- Additional other configurations are possible such as random number generation, post encrypting of shares etc.

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AoNT | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 1 | X | | | | | | | X | | | | | | |
| 2 | X | | | | | | | | X | | | | | |
| 3 | X | | | | | | | | | X | | | | |
| 4 | X | X | | | X | | | X | | | X | | | |
| 5 | X | X | | | X | | | X | | | | X | | |
| 6 | X | X | | | X | | | X | | | | X | | X |
| 7 | X | X | | | X | | | X | | | | | X | |
| 8 | X | X | | | X | | | X | | | | | X | X |
| 9 | X | X | | | X | | | | X | | X | | | |
| 10 | X | X | | | X | | | | X | | | X | | |
| 11 | X | X | | | X | | | | X | | | X | | X |
| 12 | X | X | | | X | | | | X | | | | X | |
| 13 | X | X | | | X | | | | X | | | | X | X |
| 14 | X | X | | | X | | | | | X | X | | | |
| 15 | X | X | | | X | | | | | X | | X | | |
| 16 | X | X | | | X | | | | | X | | X | | X |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AoNT | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 17 | X | X | | | X | | | | | X | | | X | |
| 18 | X | X | | | X | | | | | X | | | X | X |
| 19 | X | | X | | X | | | X | | | X | | | |
| 20 | X | | X | | X | | | X | | | | X | | |
| 21 | X | | X | | X | | | X | | | | X | | X |
| 22 | X | | X | | X | | | X | | | | | X | |
| 23 | X | | X | | X | | | X | | | | | X | X |
| 24 | X | | X | | X | | | | X | | X | | | |
| 25 | X | | X | | X | | | | X | | | X | | |
| 26 | X | | X | | X | | | | X | | | X | | X |
| 27 | X | | X | | X | | | | X | | | | X | |
| 28 | X | | X | | X | | | | X | | | | X | X |
| 29 | X | | X | | X | | | | | X | X | | | |
| 30 | X | | X | | X | | | | | X | | X | | |
| 31 | X | | X | | X | | | | | X | | X | | X |
| 32 | X | | X | | X | | | | | X | | | X | |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AoNT | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 33 | X | | X | | X | | | | | X | | | X | X |
| 34 | X | | | X | X | | | X | | | X | | | |
| 35 | X | | | X | X | | | X | | | | X | | |
| 36 | X | | | X | X | | | X | | | | X | | X |
| 37 | X | | | X | X | | | X | | | | | X | |
| 38 | X | | | X | X | | | X | | | | | X | X |
| 39 | X | | | X | X | | | | X | | X | | | |
| 40 | X | | | X | X | | | | X | | | X | | |
| 41 | X | | | X | X | | | | X | | | X | | X |
| 42 | X | | | X | X | | | | X | | | | X | |
| 43 | X | | | X | X | | | | X | | | | X | X |
| 44 | X | | | X | X | | | | | X | X | | | |
| 45 | X | | | X | X | | | | | X | | X | | |
| 46 | X | | | X | X | | | | | X | | X | | X |
| 47 | X | | | X | X | | | | | X | | | X | |
| 48 | X | | | X | X | | | | | X | | | X | X |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AoNT | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 49 | X | X | | | | X | | X | | | X | | | |
| 50 | X | X | | | | X | | X | | | | X | | |
| 51 | X | X | | | | X | | X | | | | X | | X |
| 52 | X | X | | | | X | | X | | | | | X | |
| 53 | X | X | | | | X | | X | | | | | X | X |
| 54 | X | X | | | | X | | | X | | X | | | |
| 55 | X | X | | | | X | | | X | | | X | | |
| 56 | X | X | | | | X | | | X | | | X | | X |
| 57 | X | X | | | | X | | | X | | | | X | |
| 58 | X | X | | | | X | | | X | | | | X | X |
| 59 | X | X | | | | X | | | | X | X | | | |
| 60 | X | X | | | | X | | | | X | | X | | |
| 61 | X | X | | | | X | | | | X | | X | | X |
| 62 | X | X | | | | X | | | | X | | | X | |
| 63 | X | X | | | | X | | | | X | | | X | X |
| 64 | X | | X | | | X | | X | | | X | | | |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AoNT | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 65 | X | | X | | | X | | X | | | | X | | |
| 66 | X | | X | | | X | | X | | | | X | | X |
| 67 | X | | X | | | X | | X | | | | | X | |
| 68 | X | | X | | | X | | X | | | | | X | X |
| 69 | X | | X | | | X | | | X | | X | | | |
| 70 | X | | X | | | X | | | X | | | X | | |
| 71 | X | | X | | | X | | | X | | | X | | X |
| 72 | X | | X | | | X | | | X | | | | X | |
| 73 | X | | X | | | X | | | X | | | | X | X |
| 74 | X | | X | | | X | | | | X | X | | | |
| 75 | X | | X | | | X | | | | X | | X | | |
| 76 | X | | X | | | X | | | | X | | X | | X |
| 77 | X | | X | | | X | | | | X | | | X | |
| 78 | X | | X | | | X | | | | X | | | X | X |
| 79 | X | | | X | | X | | X | | | X | | | |
| 80 | X | | | X | | X | | X | | | | X | | |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AoNT | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other External | | None | Hash | MAC | Dis Sig |
| 81 | X | | | X | | X | | X | | | | X | | X |
| 82 | X | | | X | | X | | X | | | | | X | |
| 83 | X | | | X | | X | | X | | | | | X | X |
| 84 | X | | | X | | X | | | X | | X | | | |
| 85 | X | | | X | | X | | | X | | | X | | |
| 86 | X | | | X | | X | | | X | | | X | | X |
| 87 | X | | | X | | X | | | X | | | | X | |
| 88 | X | | | X | | X | | | X | | | | X | X |
| 89 | X | | | X | | X | | | | X | X | | | |
| 90 | X | | | X | | X | | | | X | | X | | |
| 91 | X | | | X | | X | | | | X | | X | | X |
| 92 | X | | | X | | X | | | | X | | | X | |
| 93 | X | | | X | | X | | | | X | | | X | X |
| 94 | X | X | | | | | X | X | | | X | | | |
| 95 | X | X | | | | | X | X | | | | X | | |
| 96 | X | X | | | | | X | X | | | | X | | X |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AoNT | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 97 | X | X | | | | | X | X | | | | | X | |
| 98 | X | X | | | | | X | X | | | | | X | X |
| 99 | X | X | | | | | X | | X | | X | | | |
| 100 | X | X | | | | | X | | X | | | X | | |
| 101 | X | X | | | | | X | | X | | | X | | X |
| 102 | X | X | | | | | X | | X | | | | X | |
| 103 | X | X | | | | | X | | X | | | | X | X |
| 104 | X | X | | | | | X | | | X | X | | | |
| 105 | X | X | | | | | X | | | X | | X | | |
| 106 | X | X | | | | | X | | | X | | X | | X |
| 107 | X | X | | | | | X | | | X | | | X | |
| 108 | X | X | | | | | X | | | X | | | X | X |
| 109 | X | | X | | | | X | X | | | X | | | |
| 110 | X | | X | | | | X | X | | | | X | | |
| 111 | X | | X | | | | X | X | | | | X | | X |
| 112 | X | | X | | | | X | X | | | | | X | |

EX1015, Page 190

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AoNT | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 113 | X | | | X | | | X | X | | | | | X | X |
| 114 | X | | | X | | | X | | X | | X | | | |
| 115 | X | | | X | | | X | | X | | | X | | |
| 116 | X | | | X | | | X | | X | | | X | | X |
| 117 | X | | | X | | | X | | X | | | | X | |
| 118 | X | | | X | | | X | | X | | | | X | X |
| 119 | X | | | X | | | X | | | X | X | | | |
| 120 | X | | | X | | | X | | | X | | X | | |
| 121 | X | | | X | | | X | | | X | | X | | X |
| 122 | X | | | X | | | X | | | X | | | X | |
| 123 | X | | | X | | | X | | | X | | | X | X |

# SecureParser ToolKit Options

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Other Encryption | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 124 | X | | | | | | | X | | | | | | |
| 125 | X | | | | | | | | X | | | | | |
| 126 | X | | | | | | | | | X | | | | |
| 127 | X | X | | | X | | | X | | | X | | | |
| 128 | X | X | | | X | | | X | | | | X | | |
| 129 | X | X | | | X | | | X | | | | X | | X |
| 130 | X | X | | | X | | | X | | | | | X | |
| 131 | X | X | | | X | | | X | | | | | X | X |
| 132 | X | X | | | X | | | | X | | X | | | |
| 133 | X | X | | | X | | | | X | | | X | | |
| 134 | X | X | | | X | | | | X | | | X | | X |
| 135 | X | X | | | X | | | | X | | | | X | |
| 136 | X | X | | | X | | | | X | | | | X | X |
| 137 | X | X | | | X | | | | | X | X | | | |
| 138 | X | X | | | X | | | | | X | | X | | |
| 139 | X | X | | | X | | | | | X | | X | | X |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Other Encryption | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 140 | X | X | | | X | | | | | X | | | X | |
| 141 | X | X | | | X | | | | | X | | | X | X |
| 142 | X | | X | | X | | | X | | | X | | | |
| 143 | X | | X | | X | | | X | | | | X | | |
| 144 | X | | X | | X | | | X | | | | X | | X |
| 145 | X | | X | | X | | | X | | | | | X | |
| 146 | X | | X | | X | | | X | | | | | X | X |
| 147 | X | | X | | X | | | | X | | X | | | |
| 148 | X | | X | | X | | | | X | | | X | | |
| 149 | X | | X | | X | | | | X | | | X | | X |
| 150 | X | | X | | X | | | | X | | | | X | |
| 151 | X | | X | | X | | | | X | | | | X | X |
| 152 | X | | X | | X | | | | | X | X | | | |
| 153 | X | | X | | X | | | | | X | | X | | |
| 154 | X | | X | | X | | | | | X | | X | | X |
| 155 | X | | X | | X | | | | | X | | | X | |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Other Encryption | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 156 | X | | X | | X | | | | | X | | | X | X |
| 157 | X | | | X | X | | | X | | | X | | | |
| 158 | X | | | X | X | | | X | | | | X | | |
| 159 | X | | | X | X | | | X | | | | X | | X |
| 160 | X | | | X | X | | | X | | | | | X | |
| 161 | X | | | X | X | | | X | | | | | X | X |
| 162 | X | | | X | X | | | | X | | X | | | |
| 163 | X | | | X | X | | | | X | | | X | | |
| 164 | X | | | X | X | | | | X | | | X | | X |
| 165 | X | | | X | X | | | | X | | | | X | |
| 166 | X | | | X | X | | | | X | | | | X | X |
| 167 | X | | | X | X | | | | | X | X | | | |
| 168 | X | | | X | X | | | | | X | | X | | |
| 169 | X | | | X | X | | | | | X | | X | | X |
| 170 | X | | | X | X | | | | | X | | | X | |
| 171 | X | | | X | X | | | | | X | | | X | X |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Other Encryption | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 172 | X | X | | | | X | | X | | | X | | | |
| 173 | X | X | | | | X | | X | | | | X | | |
| 174 | X | X | | | | X | | X | | | | X | | X |
| 175 | X | X | | | | X | | X | | | | | X | |
| 176 | X | X | | | | X | | X | | | | | X | X |
| 177 | X | X | | | | X | | | X | | X | | | |
| 178 | X | X | | | | X | | | X | | | X | | |
| 179 | X | X | | | | X | | | X | | | X | | X |
| 180 | X | X | | | | X | | | X | | | | X | |
| 181 | X | X | | | | X | | | X | | | | X | X |
| 182 | X | X | | | | X | | | | X | X | | | |
| 183 | X | X | | | | X | | | | X | | X | | |
| 184 | X | X | | | | X | | | | X | | X | | X |
| 185 | X | X | | | | X | | | | X | | | X | |
| 186 | X | X | | | | X | | | | X | | | X | X |
| 187 | X | | X | | | X | | X | | | X | | | |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Other Encryption | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 188 | X | | X | | | X | | X | | | | X | | |
| 189 | X | | X | | | X | | X | | | | X | | X |
| 190 | X | | X | | | X | | X | | | | | X | |
| 191 | X | | X | | | X | | X | | | | | X | X |
| 192 | X | | X | | | X | | | X | | X | | | |
| 193 | X | | X | | | X | | | X | | | X | | |
| 194 | X | | X | | | X | | | X | | | X | | X |
| 195 | X | | X | | | X | | | X | | | | X | |
| 196 | X | | X | | | X | | | X | | | | X | X |
| 197 | X | | X | | | X | | | | X | X | | | |
| 198 | X | | X | | | X | | | | X | | X | | |
| 199 | X | | X | | | X | | | | X | | X | | X |
| 200 | X | | X | | | X | | | | X | | | X | |
| 201 | X | | X | | | X | | | | X | | | X | X |
| 202 | X | | | X | | X | | X | | | X | | | |
| 203 | X | | | X | | X | | X | | | | X | | |

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Other Encryption | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other External | | None | Hash | MAC | Dis Sig |
| 204 | X | | | X | | X | | X | | | | X | | X |
| 205 | X | | | X | | X | | X | | | | | X | |
| 206 | X | | | X | | X | | X | | | | | X | X |
| 207 | X | | | X | | X | | | X | | X | | | |
| 208 | X | | | X | | X | | | X | | | X | | |
| 209 | X | | | X | | X | | | X | | | X | | X |
| 210 | X | | | X | | X | | | X | | | | X | |
| 211 | X | | | X | | X | | | X | | | | X | X |
| 212 | X | | | X | | X | | | | X | X | | | |
| 213 | X | | | X | | X | | | | X | | X | | |
| 214 | X | | | X | | X | | | | X | | X | | X |
| 215 | X | | | X | | X | | | | X | | | X | |
| 216 | X | | | X | | X | | | | X | | | X | X |
| 217 | X | X | | | | | X | X | | | X | | | |
| 218 | X | X | | | | | X | X | | | | X | | |
| 219 | X | X | | | | | X | X | | | | X | | X |

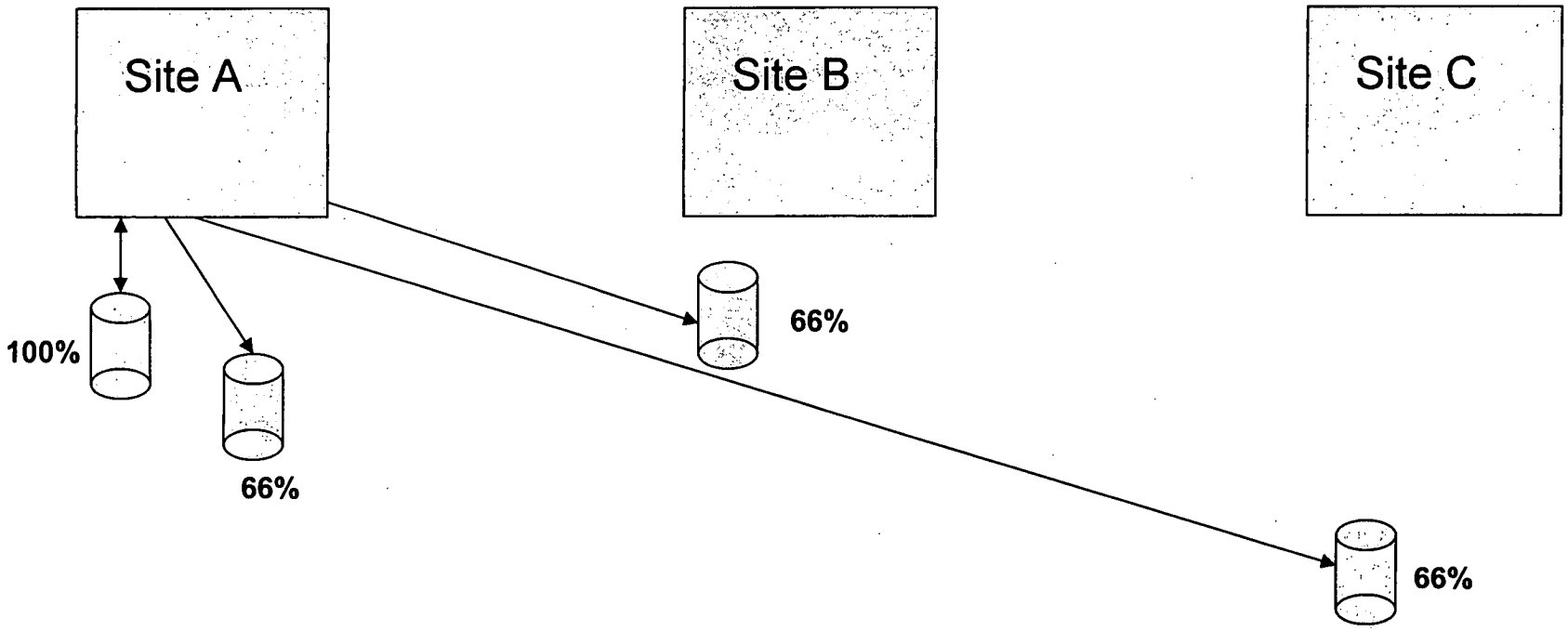
# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Other Encryption | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 220 | X | X | | | | | X | X | | | | | X | |
| 221 | X | X | | | | | X | X | | | | | X | X |
| 222 | X | X | | | | | X | | X | | X | | | |
| 223 | X | X | | | | | X | | X | | | X | | |
| 224 | X | X | | | | | X | | X | | | X | | X |
| 225 | X | X | | | | | X | | X | | | | X | |
| 226 | X | X | | | | | X | | X | | | | X | X |
| 227 | X | X | | | | | X | | | X | X | | | |
| 228 | X | X | | | | | X | | | X | | X | | |
| 229 | X | X | | | | | X | | | X | | X | | X |
| 230 | X | X | | | | | X | | | X | | | X | |
| 231 | X | X | | | | | X | | | X | | | X | X |
| 232 | X | | X | | | | X | X | | | X | | | |
| 233 | X | | X | | | | X | X | | | | X | | |
| 234 | X | | X | | | | X | X | | | | X | | X |
| 235 | X | | X | | | | X | X | | | | | X | |

# SecureParser ToolKit Options (cont'd)

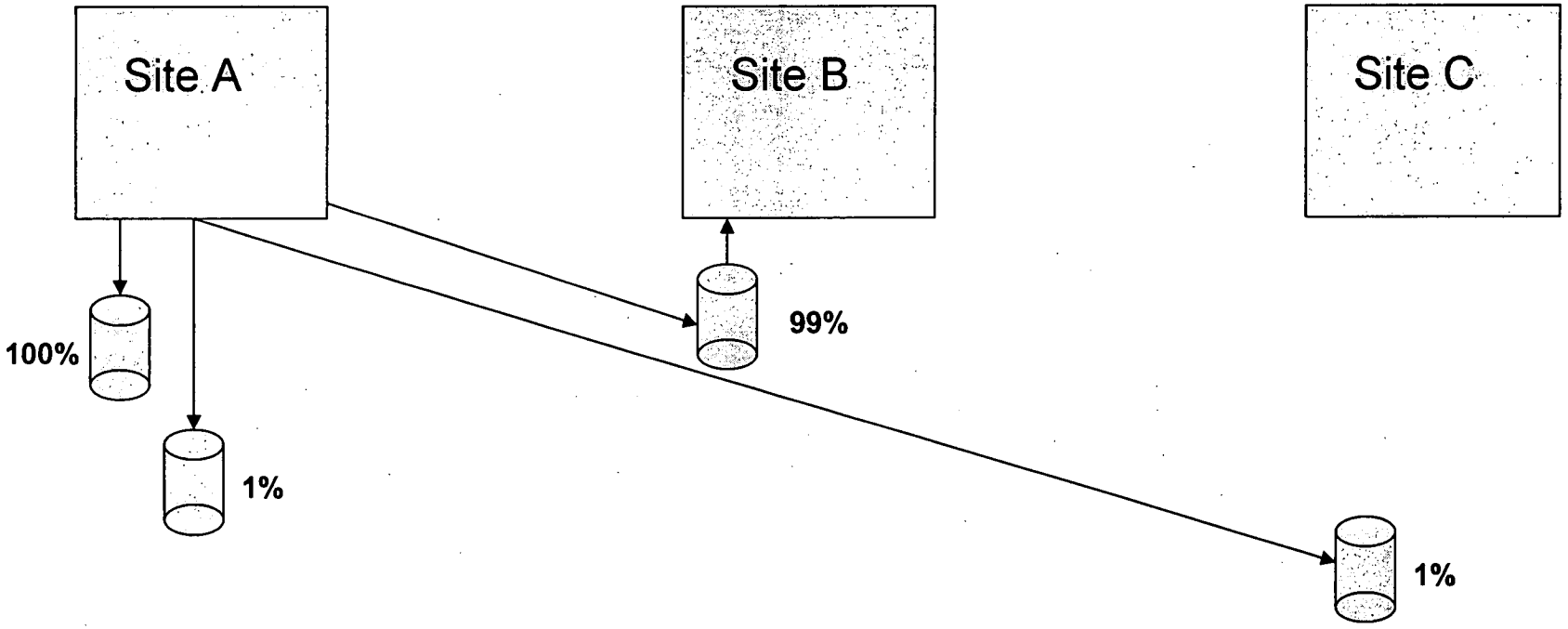
| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Other Encryption | Bit Split | Block Shuffle | Ordered Block | N of N | M of N | Parity | Shamir | Other | External | None | Hash | MAC | Dis Sig |
| 236 | X | | | X | | | X | X | | | | | X | X |
| 237 | X | | | X | | | X | | X | | X | | | |
| 238 | X | | | X | | | X | | X | | | X | | |
| 239 | X | | | X | | | X | | X | | | X | | X |
| 240 | X | | | X | | | X | | X | | | | X | |
| 241 | X | | | X | | | X | | X | | | | X | X |
| 242 | X | | | X | | | X | | | X | X | | | |
| 243 | X | | | X | | | X | | | X | | X | | |
| 244 | X | | | X | | | X | | | X | | X | | X |
| 245 | X | | | X | | | X | | | X | | | X | |
| 246 | X | | | X | | | X | | | X | | | X | X |

# SecureParser fault tolerance to 3 sites (no backup) (Phase 2 option 2, key split with data)




1. Data created locally is cleartext and local to site A
2. SecureParser secures each block written at site A
3. Data is split so that 66% is written to site A, 66% to both site B and C
4. Failover allows 100% of data to be recreated from any 2 of 3 sites

# Secure offsite replica using 3 sites (phase 2 option 1 key split with data)




1. Data created locally is cleartext and local to site A
2. SecureParser secures each block written at site A
3. Data is split so that 99% is written to site B, 1% to both site A and C
4. Failover allows 100% of data to be recreated from 99% of data from site B and 1% from either site A or C
5. SecureParser restores protected data to cleartext at site B

# PROBLEM : BACKUP TAPES & VULNERABILITIES

The security of backup tapes is of great importance due to increased regulatory attention and publicity attending the loss of sensitive data. Tape backup data represents a significant vulnerability in many security policies, as tapes frequently contain significant quantities of high-value data in a portable form. Furthermore, centralized data backup operations frequently provide a path by which sensitive data can become unintentionally mixed with less sensitive data, leading to an inappropriate deployment of physical security for some backup tapes.

We begin by briefly noting that "traditional" encryption is only one technique for providing security for backup tapes, and detail a set of disadvantages to this approach. These limitations derive primarily from the need to securely perform key management. We then describe the technique used by the SecureParser, which employs NSA Type 1 cryptographic primitives to provide an alternative approach to data security. We note that the SecureParser approach simplifies key management and obviates the need for complex key-storage hardware and training for backup operators.

## CONCERNS

### Single Point of Failure - Encryption-only approach

"Traditional" encryption encompasses symmetric and public-key encryption operations, which make use of encryption keys to encipher/decipher data. While these techniques sufficiently protect data, their practical use is complicated primarily by the need to securely perform key storage and management. This necessitates an additional set of procedures for generating, storing, and auditing the use of encryption/decryption keys. Fundamentally, an encryption-based approach is no more secure than the procedures applied to the encryption keys.

Should keys become compromised at any point before, during or subsequent to a data-loss incident, encryption offers no protection for the compromised data.

### Key Management = Extra Work & Costs

**Key Generation:** When unique encryption keys are generated for each backup operation, each new key must be stored. This can be done using a local key-store, maintained at the backup location. However, in this scenario, key material may be vulnerable to compromise along with data. This concern can be mitigated through the use of tamper-resistant hardware, key encryption, offsite key-storage, etc. However, many of these approaches serve only to shift the problem of key storage to another potential point of failure.

**Key Re-use – Exposure & Compromise:** Use of long-term encryption keys over multiple backups significantly reduces the quantity of key material that must be stored. Furthermore, when a single key is used repeatedly, its value is greatly increased and it becomes a significantly more likely target for compromise.

Simultaneously, it must be exposed more frequently, in order to facilitate tape decryption (and in some schemes, encryption as well). When using symmetric-key (secret key) encryption algorithms, a single key is used to both encrypt and decrypt documents. This allows for potential exposure of keys at tape creation and at a later decryption point. The use of Public Key cryptography can mitigate this problem by providing a separate (non-secret) key for use specifically when encrypting tapes (this key cannot be used to decrypt tapes). However, data recovery from tape still requires access to the secret decryption key, which must be conveyed from secure storage and used by some responsible party.

**Key Loss (Destruction):** Loss or destruction of encryption keys renders stored data unrecoverable. Therefore, in any approach to data backup where keys required for recovery, the system should include a separate mechanism designed to "back up" keys themselves (e.g., storing keys in separate physical locations, etc.) However, each key replica represents a single point of failure, and must be protected. Furthermore, stewardship of keys by multiple parties substantially increases the risk of insider attack. (This can be mitigated through key-splitting approaches where multiple "split" keys must be re-combined to recover an encryption keys. SecureParser provides a similar functionality using a more flexible security mechanism.)

## SOLUTION : SECUREPARSER

**Data Splitting – Parsing:** The cryptographic data-splitting approach of SecureParser addresses many of the above concerns by obviating the need for complex key management schemes. SecureParser operates by dividing input data into multiple "shares", which may be placed onto separate tapes (or on other media including networked hard drives, NV-RAM storage devices, etc.). Any party who obtains one (or fewer than a pre-specified number) of these shares will be unable to recover the original data. In an example scenario, the SecureParser might "split" a single multi-gigabyte input across a set of four backup tapes (each containing approximately ¼ of the total material), such that *all four tapes* must be recovered in order to recover the initial file. At the same

time, the SecureParser approach provides cryptographic guarantees that no 3 of the backup tapes will reveal *any* useful information about the original data.

## Advantages

**Significant Advantage:** This approach to data security has significant advantages over key-based techniques, in that it is easily deployed and can be understood by all players in a security policy. Provided that participants can prevent the loss of some (user-specified) number of shares, parsing provides an identical level of security to traditional encryption techniques. Furthermore, SecureParser can offer a protection not offered by keyed-encryption, which is a form of "security through bulk", where an attacker must recover most of a potentially very large backup tape in order to obtain useful data. This prevents insiders from memorizing and stealing secrets, as might be possible with decryption keys.

**Cryptographic Security:** The security guarantees of the SecureParser system are equivalent to those provided by traditional (keyed) encryption approaches. This is due to the fact that SecureParser is itself a form of (un-keyed) encryption, and uses an NSA Type 1 encryption algorithm (AES) as its core component. (In an alternative sense, the SecureParser can be considered as a keyed encryption algorithm where the shares themselves combine to form the key.) The constructions used by SecureParser provide the same level of security as AES against any attacker who does not possess the correct number of shares. Using standard parameters, this will prevent recovery even by an attacker with enormous resources. Moreover, the SecureParser algorithm is highly efficient, and operates at speeds similar to other secure encryption modes (on some hardware, parsing throughput may be significantly greater than many encryption techniques, due to the parallelizable algorithms used).

**Physical Separation - Data Storage and Transport Application:** In practice, SecureParser allows for flexible storage policies whereby critical data is split onto separate media (e.g., tapes), which may then be stored in disparate locations. To prevent loss of sensitive data, it is only necessary to ensure that an attacker be unable to recover a sufficient set of shares to recover the original data. By distributing tapes sensibly, an organization can ensure data confidentiality unless an attacker is able to compromise several separate storage locations (or, alternatively, obtain the collusion of a large number of insiders). Alternatively, data can be transported securely to an offsite location by splitting data into multiple shares, which are then transported via independent channels or routes (electronic or physical). Provided that at least 1 (or some pre-specified) number of transport channels are secure, data confidentiality is guaranteed.

**Resilience to Share Loss / Destruction:** Using traditional encryption approaches, data recovery depends on the availability of decryption keys. In that approach,

the loss or destruction of a decryption key will prevent any future recovery of an encrypted backup tape, rendering it useless. Protecting against key loss in traditional encryption scenarios requires complex key management techniques, which may not be appropriate for many deployments.

**Fault Tolerance:** While SecureParser shares may also be lost or destroyed, SecureParser includes built-in disaster recovery features to account for the possibility of share destruction. To allow for data recovery in the event of share destruction, SecureParser allows for the creation of "*m* of *n*" data splits. This technique produces *n* individual shares, but only requires some subset (*m*) of those shares in order to achieve data recovery. This provides a level of redundancy in the event that some shares are not available during the re-construction process. In an example scenario, critical data might be split onto four separate backup tapes. In the event that any one tape is damaged or destroyed, the remaining three tapes can be recombined to recover the original data. This "3 of 4" split is one of many possible parameterizations of the parser. In addition to its security properties, this approach can be used to add simple redundancy, as in the case of RAID drives or striped backup tapes.

**Policies & Procedures**: The "*m* of *n*"-split can also be used to create flexible security policies. For example, a backup operator might produce *n* tapes for storage by distinct individuals or departments. Based on the settings of SecureParser, any quorum of *m* individuals can combine their shares in order to re-construct data. This addresses a recovery scenario where only certain users are available, but also prevents insider attack by some smaller number of users.

**Mandatory Share - Advanced Options and Secure Deletion:** To provide additional flexibility, SecureParser can also provide for options such as "mandatory shares" which will *always* be required in order to reconstruct a document. These shares serve as a backstop—in a sense, a "master key"—without which no set of shares can be used to reconstruct the document. This has useful implications for tape re-purposing and destruction. Once a mandatory share has been destroyed, the entire underlying document is effectively destroyed (the remaining shares are will not allow for data recovery). This allows for rapid destruction of data once it is no longer required for storage, and can allow for re-purposing of the remaining tapes or storage devices.

**Efficiency:** The SecureParser algorithms process data at rates similar or greater to other (secure) encryption schemes. SecureParser's AES-based algorithm is optimal, i.e. operates at the same rate as other secure AES-encryption approaches. (Although the SecureParser can be adapted to use other ciphers, AES is recommended due to its security and efficiency.) The SecureParser algorithm offers a set of internal settings, which adjust parameters such as the AES cipher's key length (128/192/256-bit). These decisions can enhance the security level of parsed shares, at some cost to performance.

**Throughput Improvement – MultiPathing:** Moreover, the encryption schemes used by SecureParser are explicitly designed to take advantage of advanced hardware features such as multi-processing and dedicated cryptographic hardware, in order to multiply data throughput. SecureParser algorithms use a non-sequential encryption mode (AES-CTR) so as to allow parallel processing of data, which allows for many times speedup in the event that these features are available in the underlying hardware.

**Faster – Elapsed Time Decreased:** In a common use of SecureParser, backup data shares are written to separate media (e.g., multiple tapes). In these cases, SecureParser allows for faster backup via parallel writes. Customers who possess multiple tape drives can perform this splitting and media-write process on several drives simultaneously. For instance, a customer might divide input data into 8 shares, which could be simultaneously written to 8 separate tapes. Consequently, a large backup job can be performed in a fraction of the time required to write data one tape at a time.

**Integrity of Data / Authentication:** A concern raised in the referenced document is the possibility of data tampering, e.g., the placement of a Trojan Horse program onto a stored backup. Such tampering can offer a back-channel by which malicious code or data can be inserted into a secure system. SecureParser offers built-in integrity and authentication for shares based on Message Authentication Codes and Digital Signatures (e.g., HMAC-SHA1 and RSA-PSS) . This approach provides cryptographic guarantees against data modification by malicious actors.

## The SecureParser Data Processing Module

The SecureParser is a cryptographic software module designed to efficiently process full-duplex data streams with specific functionality for IPv4/IPv6 network traffic and other common network protocols. The SecureParser ensures the confidentiality of transmitted data using industry standard strong data encryption techniques and algorithms, while adding minimal latency and maximizing performance on a variety of single- and multi-processing architectures.

In addition to providing FIPS-compliant encryption techniques, the SecureParser implements a novel cryptographic data splitting technique designed to transparently multiplex data-streams over multiple sub-channels, including wired and wireless connections. This process allows transmitters to transparently combine several channels into a single high-bandwidth "virtual" channel without introducing significant latency to impede real-time communications applications. This innovative data-splitting technology provides additional security guarantees above and beyond those provided by standard encryption techniques.

The SecureParser consists of two primary components:

1. A highly-efficient encryption module designed to meet FIPS-140-02 standards for a software cryptographic module, including standard data authentication, key management and user authentication mechanisms.
2. A proprietary "data splitting" core designed to securely multiplex data-streams, with minimal latency and strong cryptographic guarantees.

The remainder of this document describes the SecureParser and discusses the potential applications of this software.

### Secure Authenticated Encryption

The SecureParser is designed to encrypt arbitrary data streams using industry-standard AES encryption and FIPS-compliant data authentication mechanisms. SecureParser provides low-latency, high-performance encryption of arbitrary data streams, and includes specific operating modes designed to secure a variety of specific data formats, including IP (v4/v6) traffic. SecureParser is capable of encapsulating encrypted output into a variety of formats for transmission over arbitrary communication channels.

**High Throughput.** The SecureParser incorporates a highly efficient AES encryption engine which complies with FIPS-140-02 requirements and provides encryption at key lengths of 128, 192 and 256 bits. The encryption algorithms included in SecureParser are implemented so as to provide optimal throughput on parallel- and multi-processor architectures, using parallelizable cipher modes wherever possible.

**Cryptographic Message Authentication and Integrity.** Data streams transmitted through the SecureParser system are authenticated using efficient FIPS-compliant

message authentication codes (HMAC-SHA1). Additionally, SecureParser incorporates digital signatures (EC-DSA and RSASSA-PSSA) for transmissions which require non-reputability and to perform authenticated key-agreement.

**Key Management and Key Agreement.** The SecureParser provides a variety of techniques for managing keys, compliant with FIPS standards for a software encryption module. These include systems for encrypting keys data, use of protected memory for key material, and "split-knowledge" techniques which allow keys to be safely divided among multiple participants.

Additionally, the SecureParser implements provably-secure authenticated Key Agreement techniques designed to rapidly establish connections between SecureParser modules, and is compliant with industry standards including IPSEC, TLS and SSL.

## Cryptographic Data Splitting

A principal feature of the SecureParser is a proprietary data-splitting technology designed to "split" or multiplex arbitrary data-streams over multiple sub-streams. This capability can be used to combine a series of low-bandwidth channels into a single high-bandwidth data channel, without introducing significant latency. Alternatively, it can be used to combine several variable-bandwidth channels into a single virtual channel with a more stable aggregate bandwidth.

The SecureParser data-splitting process also provides an additional layer of cryptographic security for transmitted data. This approach provides provable guarantees that an eavesdropper who intercepts one (or a subset) of the sub-streams will be unable to reconstruct the original data. When combined with the standard encryption protections provided by SecureParser, this approach can offer significantly enhanced security for transmitted data, at minimal computational cost.

**Low latency.** The proprietary SecureParser data-splitting technology multiplexes data at the "bit" level, rather than at the frame or packet level. This results in minimal added latency, which is critical for real-time communications protocols and voice communications.

**Cryptographic security.** To accomplish secure data splitting, the SecureParser uses a proprietary coding technology which incorporates provably-secure cryptographic elements. The security guarantees provided by this process ensure that channel re-construction is only possible when a sufficient number of sub-channels are available to the receiver. In practice, this protects the data-stream against eavesdroppers who are capable of intercepting only certain channels (e.g., broadcast channels) but not others, including wired connections and other restricted forms of radio broadcast (e.g., cellular communications). This additional layer of protection can provide a critical additional protection, particularly in the event that an encryption key is compromised.

The data-splitting approach can also be used to increase the efficiency of SecureParser or another encryption product. Because the splitting process provides provably guarantees for the confidentiality of a full data stream when some channels are unavailable to an attacker, users can protect data by encrypting only a subset of the resulting sub-streams. Due to the security properties of the splitting algorithm, the data-stream remains secure even if no encryption is applied to the remaining channels. As these encrypted sub-channels may carry significantly less data than the original data-stream, this approach can significantly reduce the burden on an encryption device. Sub-stream encryption can be used to replace or augment the protection provided by the primary SecureParser encryption functionality.

**Integrity and disaster recovery.** The SecureParser splitting algorithm may be parameterized to add redundancy to split data. In the case that some channels are unreliable, for example, SecureParser can split data into $n$ sub-streams such that any $m$ of those streams will allow for full data recovery. This ensures that data transmission is uninterrupted even in the event of temporary (or permanent) disconnection of some channels. Additionally, data may be split so that certain sub-streams are "mandatory" for re-construction of the signal, while others are not.

SecureParser can also add redundancy to individual data streams by embedding standard error correction codes, along with cryptographic integrity protections to prevent malicious tampering.

# Secured Storage Using SecureParser™

## ABSTRACT

Securing storage data is a manifold problem with requirements in three dimensions: data security, data integrity, and the safety of data. Meeting the requirements for one dimension often means compromising another. SecureParser™ is a software technology which addresses all three dimensions of secure storage without compromising any. In this paper, we describe the SecureParser™ technology and discuss how it addresses the three dimensions of secured storage: security, integrity, and safety.

## 1. INTRODUCTION

The SecureParser™ from Security First Corp. is a unique cross-platform software module suite that not only incorporates NSA cryptographic primitives providing an alternative approach to data security, but also has the ability to provide fault tolerance, increased access speed and physical separation of data. This approach eliminates many of the problems typically associated with security solutions including key management single points of failure and data unavailability.

The cryptographic data-splitting approach of the SecureParser operates by randomly dividing the input data at the bit level into multiple "shares". These shares may be stored in separate locations or transmitted across different communication channels. Any party who obtains or holds one (or less than a pre-specified number) of these shares will be unable to recover the original data. There are several methods that the SecureParser™ employs for securing and splitting data. One of these methods is discussed in detail in this paper.

The SecureParser allows for additional information to be written to each share, thereby creating a fault tolerant environment that can sustain the loss of one or more shares. For example, the output may be configured to allow for any 3 of 4 shares to recreate the original information. Each share is authenticated during creation and has built-in integrity checks to protect against intentional and unintentional damage.

The SecureParser™ approach simplifies key management and eliminates the need for complex key storage systems. For those installations where keyed encryption type systems are mandated, the SecureParser™ can be configured to pass a session key back to be stored in a key store. The security guarantees of the SecureParser™ system are similar to those provided by traditional encryption approaches. This is due to the fact that the SecureParser™ is itself a form of un-keyed encryption and uses an NSA registered encryption algorithm (AES) as its core security component. In an alternative sense, the SecureParser™ can be considered a "keyed" encryption algorithm where the shares themselves form the key. The construct used by the SecureParser™ guarantees that the "parsed" output possesses the security properties of the encryption algorithm, which prevents recovery even by an attacker with enormous resources. With the additional attributes of physical separation and fault tolerance, a data security level distinct from traditional, serial encryption is now available.

## 2. PROBLEM STATEMENT

To ensure the quality of data, information management systems must guarantee the safety, integrity, and security of that data.

The term "data safety" in the context of this paper refers to the requirement that data can not be lost through accident or attack. Typically, rigorous backup procedures are implemented to periodically copy all data to offsite, geographically dispersed, locations. Of course, there are windows in time when data is at risk, specifically between when a file is modified and when it is backed up. Mirroring technologies help close the timing holes, but at a high cost. In general, ensuring data safety in traditional ways incurs high costs in manpower and equipment.

"Data integrity" means that data can not be changed, either through accident or by design. Enterprise-class IT equipment protects against data corruption with hardware mechanisms such as parity and error-correcting codes. More insidious, however, are malicious modifications that go undetected until they have caused system crashes or erroneous decisions. Checksumming and other techniques such as digital watermarking may be employed by software to guarantee the integrity of data files, but those techniques are processor intensive and are often not applied consistently across and enterprise's digital assets.

Finally, "data security" means that data can not be read or copied by unauthorized agents. Maintaining safety and security of data often means working at cross purposes. To be safe, data should be duplicated as many times as necessary to ensure a copy will survive the worst case disaster. This could very well mean multiple, remote backups and/or replication of live data to disaster recovery sites. On the flip side, to be secure, data should exist in only one place, since the more data is replicated, the more likely it is to be stolen.

SecureParser™ and the methods by which it may be employed, as described herein, can maximize all three aspects of data safety, integrity, and security in a coherent manner.

### 2.1 Solving the Data Security Requirement

"Traditional" encryption employ private or public-key encryption operations, which make use of encryption keys to encipher/decipher data in a serial or streamed fashion. The practical use of these techniques is complicated primarily by the need to securely perform key storage and key management. This necessitates an additional set of procedures for generating, storing, and auditing the use of encryption/decryption keys. Fundamentally, an encryption based approach is no more secure than the procedures applied to the encryption keys. Should keys become compromised before, during or subsequent to a data loss

incident, encryption offers no protection for the compromised data.

The SecureParser[tm] has advantages in performance over other security solutions. The encryption operations are efficient and operate on small blocks of data. The algorithm is explicitly designed to take advantage of advanced hardware features such as multi-processing and dedicated cryptographic hardware in order to multiply data throughput performance. A 'parallelizable' encryption mode (AES-CTR) is used so as to allow for parallel processing of data, which allows for significant speed enhancement if those features are available in the underlying hardware. Also, since the output of the SecureParser[tm] is multiple shares, each smaller than the original data, multiple channels to storage arrays or communication channels allow for multi-threaded I/O, enhancing speed of access during read and write operations.

The SecureParser[tm] is parameter based, allowing for configuration of parameters such as the number of output shares, number of shares required to recombine the data, AES key length (128/192/256 bit) and designation of mandatory shares. The SecureParser[tm] can integrate easily with PKI, securing any individual key value or share with a public key, and allowing only the owner of the private key the ability to restore the data.

The following diagram (Figure 1) outlines the functional flow of the SecureParser[tm] process in more detail.

1. Split input data (x) into "n" number of blocks of preconfigured sizes 128, 192 or 256 bits each in length. The resulting blocks are designated in the diagram as $x_1$ through $x_n$.
2. Generate a random value "k" to be used as the transform key. The random number can be generated from hardware if necessary.
3. Transform each "x" block into a corresponding "y" block using the key value "k".
4. Compute a new y value "$y_0$" to represent the XOR of all available "y" blocks and the key value "k": $y_0 = y_1$ XOR $y_2$ XOR $y_3$ ... $y_n$ XOR k.
5. Discard key value "k".
6. If a session key needs to be held, any y value, including $y_0$, may be optionally held as a session key or encrypted using a public key.
7. Resulting y values including $y_0$ are distributed randomly at the bit level to shares adding fault tolerance as specified.

To restore the information, all available bits from the shares representing the original "y" blocks must be presented

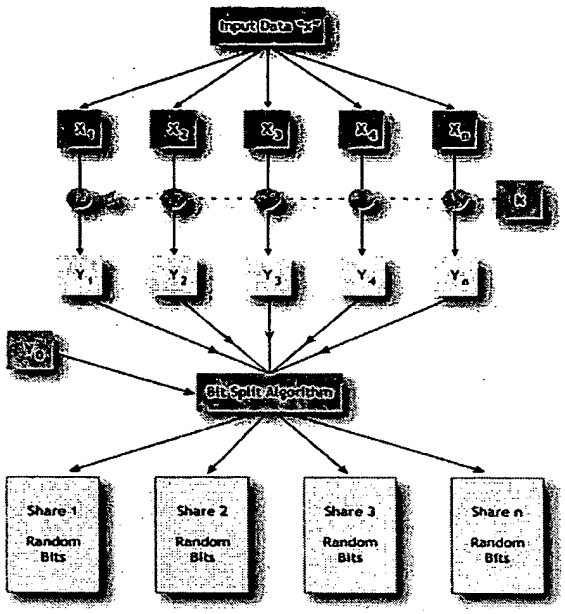



**Figure 1. SecureParser[tm] Functional Flow**

# 3. SECUREPARSER[TM] TECHNOLOGY

## 3.1 Description of Method

The SecureParser[tm] Core (Figure 2) isolates the functions of data splitting (share creation, redundancy, and distribution to share buffers) into a callable module. Data is placed in the Assembled Data Buffer to be split and the SecureParser[tm] is invoked. The resulting shares are placed into the Split Shares Buffers to be securely stored or transmitted by the host application.

All cryptographic functions are external calls for the SecureParser[tm]. This allows for the custom integration of various cryptographic libraries, giving flexibility for "bolting on" other encryption modules, key management systems, random number generators, etc. as needed.

External functions called by the SecureParser[tm] are:

- Random Number Generator – For each split operation, a unique random value is generated to be used as the basis for the split. Random numbers can be generated in a number of ways, via software program or hardware interface.
- AES Counter Mode Module – This module secures each block of data prior to the distribution to the shares.
- Encryption Algorithm – In some circumstances the data must be pre-encrypted. If this is the case, the SecureParser can invoke external encryption to process the data prior to the split. Alternatively, shares may be encrypted post-splitting.
- Key Management System – In some cases, external key management will be mandated. If so, the SecureParser will pass a key value to an external system for secure

storage.

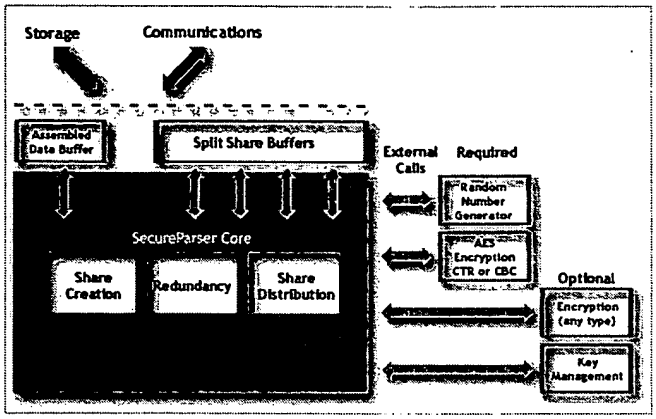



**Figure 2. SecureParser[tm] Core**

Integration to the SecureParser[tm] is accomplished through a well documented Application Programming Interface (API). Since the SecureParser[tm] is designed to be integrated to any application or infrastructure component which stores or transmits data, there are no references within the SecureParser to where the data is stored or transmitted. Data is placed into the inbound SecureParser[tm] buffer to be split, and the resulting split shares are picked up by the host application. In many cases, an API integration layer is an effective way to reduce the amount of coding needed for a particular environment. The API integration layer defines the location of the resulting shares, such as paths to send the information or storage device to store the shares. The API integration layer also defines the naming convention of the resulting data, or parameters to store with the information such as row, column and table ID in the case of a database application or the original filename in the case of unstructured data such as documents, images, maps, multimedia, or streaming data. The diagram below (Figure 3) shows the API integration layer. Overall, this API layer handles streaming of the data to the SecureParser and secure storage or communications of the resulting shares.

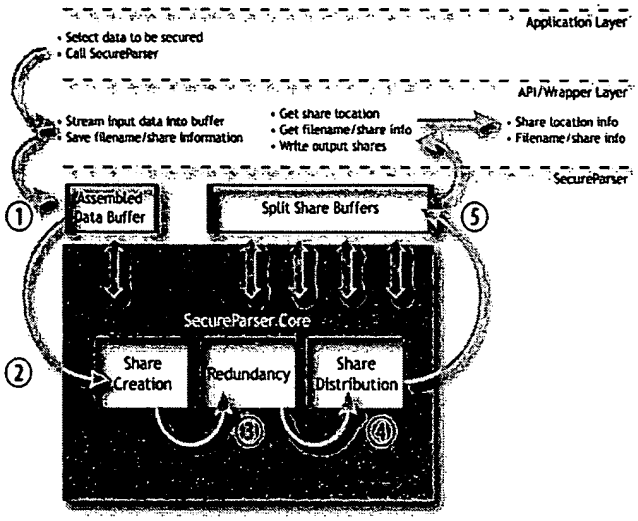



**Figure 3. SecureParser[tm] API Integration Layers**

## 3.2 Integration Points

There are several points within the data communications architecture where the SecureParser[tm] technology can be integrated. These integration points and their pros are shown in Table 1.

**Table 1. SecureParser[tm] Integration Pros**

| Layer | Pros | |
|---|---|---|
| Application | Custom integration for application differentiation and value-add. (Currently packaged with installation tools.) | |
| File system | Generic file access;<br>Local and remote files supported;<br>Share/key mapping integrated into directory;<br>Concurrent multiple paths | |
| Device driver | LUN-based rather than file-based;<br>Concurrent multiple paths;<br>Supports legacy HBAs and fabrics | |
| HBA | Offloads parsing overhead from host CPU;<br>Legacy fabrics supported | |
| Switch | Parsing is transparent to hosts;<br>Highest performance | |

An example of a Parsed File System is outlined below. Because implementation of SecureParser[tm] at the device driver and/or HBA

layers requires hardware and/or system software updates to all servers (as does a PFS), these approaches are less practical than integrating SecureParser™ into the fabric itself by implementing it within the switches.

In the sections below, the Parsed File System and Parsed Fabric options are examined.

## 3.3 Parsed File System

The Parsed File System (PFS) is a distributed file system which incorporates a directory, shared among the file servers, which is itself parsed. The PFS directory is the clients' view of all file assets. The set of files visible to an individual client is controlled by that client's group keys. An example is given in the next section.

PFS features include:

1) The distribution of SecureParser™ shares is hidden from the client. The file server location and the names of the files containing the share data (i.e. the URLs of the shares) are maintained within the directory. The file system client is given access to those shares at file open.
2) In addition to the normal file attributes such as timestamps, size, access privileges, etc., the directory stores:
   a. N-of-M
   b. URLs or some other globally unique identifier (GUID) of the shares
   c. A Key-Id referencing the actual SecureParser™ key which was used to parse the file. The file server verifies at open time that the client has been granted the key referenced by the Key-Id.
   d. Validity check data
3) The directory data is parsed with separate, replaceable group keys to allow keys to be rescinded and replenished as required, for example when personnel changes or suspected security breaches warrant.
4) Open and close operations are audited and any suspected declassifications (movement of data from a higher security level to a lower security level) are sequestered for authorization. The sequestration method is as follows:
   a. As data is read from a file, a signature of the data is calculated every S bytes (S is a tunable parameter).
   b. Signatures, along with the file's group key are stored by the file server until all of the following are true:
      i. the file is closed
      ii. all processes running when the file was opened are terminated
      iii. all processes which were started after the file was opened but while there was data in the clipboard are terminated
   c. Target signatures are generated on writes and compared to read signatures of files with different group keys to detect suspicious cut-and-paste operations.
5) Files parsed with group keys not included in the user's authorized key set are not visible to that user in the directory.
6) Signature generators for different file types (e.g. .TXT, .WRD, .XLS, .JPG) can be added to PFS through a plug-in interface.
7) Group keys can be defined to inherit privilege from other keys. For example:
   a. Unclassified = {UNCLASS}
   b. Secret = {SECRET, UNCLASS}
   c. TS/SCI = {TS_SCI, SECRET, UNCLASS}
   d. Special1 = {SAP1, SECRET, UNCLASS}
   e. Special2 = {SAP2, TS_SCI, SECRET, UNCLASS}
   f. Special3 = {SAP3, SAP1, TS_SCI, SECRET, UNCLASS}

### *3.3.1 PFS Example*

A sample use-case illustrates the PFS behavior.

1) USER1 logs in to the network and is granted {USER1, SAP1, SECRET, UNCLASS} keys.
2) USER1 opens Notepad and types "The quick fox jumped over the lazy brown dog."
3) USER1 saves the file as TEST1.TXT.
4) A dialog box appears asking what SecureParser™ group key should be used: USER1, SAP1, SECRET, or UNCLASS.
5) USER1 chooses SECRET and TEST1.TXT is saved.
6) USER2 logs in and is granted {USER2, UNCLASS} keys.
7) TEST1.TXT is not visible when USER2 opens Explorer.
8) USER3 logs in and is granted {USER3, TS_SCI, SAP1, SECRET, UNCLASS} keys.
9) USER3 sees TEST1.TXT and opens it.
10) USER3 copies "quick fox" to clipboard.
11) USER3 pastes clipboard ("quick fox") to a new file.
12) USER3 saves the new file as TEST2.TXT and picks the UNCLASS group key.
13) A warning dialog box appears with the message "TEST2.TXT will be sequestered for declassification review."
14) USER3 cancels the warning – the new file is not saved.
15) USER3 saves TEST2.TXT using SAP1 key. Since SAP1 includes SECRET, which was TEST1.TXT's key, no warning is issued.
16) USER1 sees and reads TEST2.TXT (since he has the SAP1 key), but USER2 can not.
17) USER3 copies/pastes "brown dog" to a new file and saves the new file as TEST3.TXT using key UNCLASS.
18) The declassification warning appears and USER3 OKs it.
19) ADMIN logs in and is granted all keys.
20) ADMIN receives a message to review TEST3.TXT for possible declassification.
21) ADMIN shows Details>>>
    File TEST3.TXT, written by USER3 using key UNCLASS. It contains "brown dog", which may have been copied from TEST1.TXT (key SECRET).
22) ADMIN OKs the save.
23) USER1, USER2, and USER3 all see and can access TEST3.TXT.

## 3.4 Parsed Fabric

An alternative to the Parsed File System is an approach which pushes the parsing operation into the storage fabric. Rather than parsing individual files, specific logical unit numbers (LUNs), i.e. drives, are parsed.

In the method shown above, the host application and server have no involvement in the parsing process. They simply issue standard operations (Read, Write, etc.) into the fabric. The Parser-enabled Switch is responsible for accessing the Key Manager and a LUN Map, somewhere in the fabric, which

contains a mapping from application-visible LUNs to physical locations of the corresponding shares.

By pushing the parsing function into the fabric, all parsing processing is offloaded from the servers. More importantly, no changes are required to application, device driver, or operating system software. In environments where rigorous certification processes are used to qualify all software, being host-transparent is a significant advantage.

Of course, the Parser-enabled Switch must be certified to interoperate with all of the appropriate fabric and storage equipment.

## 4. Differentiators

There are several key advantages of the SecureParser[tm] over typical encryption methods:

- The SecureParser[tm] offers keyless security, eliminating the complexity of generation, storage and long term management of encryption keys.
- Physical separation of data allows for flexible data storage and transmission policies which are unable to be achieved using existing encryption.
- The fault tolerant mode of the SecureParser[tm] can allow for faster access to data, disaster recovery using the remaining shares, and can eliminate the need for data backup.
- The system is easily deployed and can be understood by all players in a security policy.
- The SecureParser[tm] offers improved speed of encryption through a parallelizable encryption mode, and faster I/O operations through multi-threading of read and write operations using smaller shares of data than the original.

### 4.1 Why Is It Secure?

The SecureParser[tm] employs strong NSA registered cryptographic primitives as its core security component. The construction used by the SecureParser[tm] guarantees that the parsed output possesses the security properties of the encryption algorithm. The SecureParser[tm] not only employs strong NSA registered cryptography at its core, but uses bit-level physical separation of all the data as a construct to improve security over typical encryption solutions, eliminating single points of failure.

### 4.2 Data Recovery

The SecureParser[tm] was designed with the ability to be deployed without the need for traditional backup practices. Offsite storage of backups, redundant systems, mirrors and replicated data create an additional security risk by exposing the entirety of the data in other locations and formats. A built in fault tolerance option in the SecureParser[tm] allows for additional information to be written to each resulting share, allowing for the recreation of the original information from less than the total number of shares. For example, using a 2 of 4 distribution model, only 2 shares would be needed to recreate the original data. This allows for the recreation of a lost or damaged share by any 2 of the remaining 3 shares and may eliminate the need for backup or replication of data.

Integrity of the individual shares is assured by the use of an authentication value embedded in the individual shares. This integrity value ensures that the share has the required information to recreate the original data and has not been tampered with or damaged. Use of a 'utility' program module to verify integrity will proactively validate the information and send an alert if a share has been damaged in any manner.

### 4.3 Key Management

The SecureParser[tm] method allows for keyless information security, eliminating the complexities of current keyed encryption solutions. Discussed below are some issues associated with key management.

#### *4.3.1 Key Generation*

When unique encryption keys are generated for each operation, each new key must be stored. This can be done using a local key-store. However, in this scenario, key material may be vulnerable to compromise along with the data. This concern can be mitigated through the use of tamper-resistant hardware, key encryption, offsite key storage, etc. However, many of these approaches serve only to shift the problem of key storage to another potential single point-of-failure.

#### *4.3.2 Key Exposure*

When using symmetric-key encryption algorithms, a single key is used to both encrypt and decrypt data. This allows for potential exposure of keys at encryption and at a later decryption point. The use of public key cryptography can mitigate this problem by providing a separate non-secret key for use specifically when encrypting data (this key cannot be used to decrypt the data). However, data recovery still requires access to the secret decryption key, which must be conveyed from a secure location and used by a responsible party.

#### *4.3.3 Key Loss*

Loss or destruction of encryption keys renders stored data unrecoverable. Therefore, in any approach to data backup where keys are required for recovery, the system should include a separate mechanism designed to "back-up" keys themselves (e.g., storing keys in a separate physical location). However, each key replica represents a single point of failure and must be protected. Furthermore, stewardship of keys by multiple parties substantially increases the risk of insider attack. This can be mitigated through key-splitting approaches where multiple "split" keys must be re-combined to recover an encryption key. The SecureParser[tm] provides this functionality using a more flexible security mechanism, similar to the two man rule for missile launching.

## 5. SUMMARY

The SecureParser[tm] offers unmatched flexibility in information security and access. At the core, strong NSA cryptographic primitives protect the data in a keyless encryption offering, eliminating the single point of failure in other data security solutions. Physical separation allows for enhanced protection of data, survivability and business continuity, as well as potential performance increases in multi-threaded read/write operations. The SecureParser's[tm] "m of n" fault tolerance mode offers improved access speed, assembling the data when the first "m"

shares are retrieved. This mode also allows for reconstruction of any share from the remaining shares in case of an intentional or unintentional failure. The mandatory share option allows for repurposing of storage media by the simple destruction of a single block of data.

The SecureParser™ is small, efficient, and portable and may be run in any number of environments, from servers to desktops, cell phones to PDAs, and dedicated appliances to smartcards.

# SecureParser – How it Effectively Tackles Customer Financial Data Theft & Storage Vulnerabilities

**Executive Summary:**

Identity Theft and Fraud is rampant and has reached epidemic proportions to become the fastest growing crime in the US. All of the solutions that have been developed are focused on minimizing breaches and losses to business while keeping implementation costs in check. Most standards require encryption of data for protection, which is expensive to implement and manage due to the complexity of securely generating and managing encryption keys. As a result, its adoption has been slow.

SecureParser provides a robust data protection solution that is better than just encryption while economical to implement and manage. SecureParser provides "data splitting" technology while offering simplified "traditional" strong encryption with flexible key management techniques, including key generation, storage and distribution facilities. However, the most notable advantage of the SecureParser product is a new form of cryptographic "data splitting", which provides robust security even in the event of key theft, or other failures in key management. Data splitting technology divides sensitive data into portions (data shares) and stores it in separate locations, such that the file data cannot be recovered without an attacker obtaining *all* (or some pre-specified number) of the resulting shares. This innovative approach to data security complements, and, in most cases, obviates the need for, complex key management techniques based.

Therefore, as an alternative to traditional encryption the SecureParser, provides a security and business continuity suite which incorporates strong AES encryption with embedded key management.

The SecureParser offers the following advantages over traditional encryption solutions:

- Integrated key storage eliminates the need for complicated and costly key management policies and procedures.
- Cryptographic data splitting creates variable "shares" allowing for physical or virtual separation of data.
- Optional additional fault tolerant data written to each share allows for recreation of original information from less than the total number of shares.
- Data cannot be restored unless all (or a predetermined number) of shares are retrieved.
- Embedded authentication information ensures the integrity of shares without exposing the information.

- Encryption keys are protected within the data, allowing for flexible security policies which are easily implemented and understood by all participants in a security policy.

The SecureParser is packaged for easy integration to existing or new applications. Database applications are supported through easily adapted field level security which secure data at point of insert into the database and supports secure access to data during select, change and delete operations. Once deployed, the SecureParser is transparent to the user and does not affect the user interface.

**Background:**

Compromise of customer data has serious consequences for financial service institutions. Theft of customer information such as customer identities or credit card or other account information can expose institutions to significant direct and indirect financial costs. Direct costs include loss due to fraud, as well as the cost of replacing stolen information, e.g., re-issuing credit cards. Indirect costs include legal, brand and public relations exposure due to lawsuit and regulatory action. Laws enacted in California demand customer notification when sensitive data is lost, and legislative efforts in other states may follow suit to implement such policies.

Even though data theft is not a new threat, over time the costs associated with information theft have risen exponentially in step with the capabilities of information thieves. The increasing sophistication of these attacks is illustrated by the recent and highly publicized breach at CardSystems Solutions Inc., in which approximately 40 million credit card numbers were stolen by thieves using a remotely activated "trojan horse" program. While the CardSystems breach is noteworthy for its complexity and for the magnitude of the loss, more traditional forms of data theft such as theft by authorized insiders is a common occurrence.

One result of the CardSystems breach has been the adoption of more rigorous security programs by Card Associations, processors and other financial institutions. Mastercard International Inc. and Visa U.S.A Inc. have jointly pushed the *Payment Card Industry Data Security Standard* (PCI) for adoption by processors and organizations which regularly deal with customer data, such as card numbers, Social Security numbers, and other customer-identifying information. The PCI standard includes a variety of policy and technical measures designed to enhance network and storage security. In particular, the PCI recommends encryption of stored data to protect it in the event that other layers of security fail. Unfortunately, implementing the PCI recommendations presents a number of challenges for organizations, due to the complexity of securely generating and managing encryption keys and systems. In the final analysis, these concerns are the most expensive and complex component of any

storage-based encryption system, and represent the most likely cause of storage vulnerabilities.

In the next sections we address the goals of a secure storage solution, and describe many of the security failures that may occur due to poor key management used in traditional encryption solutions. We then describe the advantages of the SecureParser product and explain how it can prevent data losses caused by remote and insider attacks.

## SECURE FILE STORAGE: GOALS

Before considering the appropriateness of any given approach, it is important to fully understand the requirements that a secure storage device or application must achieve. These include:

- **Confidentiality of stored data.** Any data stored to disk or tape is vulnerable to theft by a motivated attacker. A secure storage scheme must provide strong cryptographic guarantees that unauthorized users cannot derive any significant information about the protected data, and must prevent attackers—including authorized insiders—from bypassing encryption through sophisticated attacks on supporting components, including key-stores and encryption devices.
- **Integrity of stored data.** Data committed to disk or backup tape is vulnerable to modification by insiders or successful remote attackers. This can range from alteration of financial information, to insertion of malicious software (e.g., a Trojan horse). To protect against this, an encryption system must incorporate mechanisms guaranteeing that unauthorized modifications will be detected before data is used.
- **Disaster recovery.** Hardware failure or human error can destroy data or encryption keys required to recover data. In the event that a decryption key is permanently destroyed, large quantities of encrypted data may become unrecoverable. This induces substantial complexity into a key management and storage process, requiring companies to back up data *and* encryption keys to insure against loss of critical data.
- **Secure deletion.** The PCI standard recommends that institutions rapidly destroy sensitive data once it is no longer needed. However, sophisticated techniques for recovering data stored on magnetic media require time-consuming deletion processes (or costly destruction of media). A data storage solution should offer efficient and secure techniques for destroying data once it is no longer required.

## DATA ENCRYPTION: Challenges

Despite its obvious benefits in securing stored data, data encryption has seen slow adoption in many organizations. To a large extent this is a result of the complex policies required to generate and manage the keys required by the encryption process. In the following sections we briefly outline these challenges, and describe how the SecureParser's data splitting technology can address these problems by complementing or eliminating complex key management.

## Single Point of Failure - Encryption-only approach

The goal of an encryption system is to protect the confidentiality of sensitive data, even when such data falls into the hands of an attacker. "Traditional" encryption approaches provide this confidentiality by enciphering data under an *encryption key*. This key, or a corresponding *decryption key* is required in order to decipher the data. Modern cryptographic algorithms guarantee that even a resourceful, motivated attacker is unable to decipher encrypted data without access to the required key material. As a result, attackers have turned their attention to compromising decryption keys themselves. This necessitates an additional set of procedures for generating, storing, and auditing the use of encryption/decryption keys. Regardless of the technology used, a traditional encryption approach is as secure as the procedures applied to the keys. Should keys become compromised at any point before, during or subsequent to a data-loss incident, encryption offers no protection for the compromised data.

## Key Management = Extra Work & Costs

**Key Generation:** One concern with key-based approaches is the procedure used to generate keys. Are fresh keys generated with each file stored, or is a single key re-used to encrypt many files? This has significant implications for storage: when unique encryption keys are generated for each file stored, each new key must itself be securely stored. This can be done using a local key-store, maintained at the backup location. However, in this scenario, key material may be vulnerable to compromise along with data. Some specialized approaches attempt to mitigate this concern through the use of tamper-resistant hardware, key encryption, offsite key-storage, etc. However, many of these approaches serve only to shift the problem of key storage to another potential point of failure.

**Key Re-use – Exposure & Compromise:** Use of long-term encryption keys over multiple files significantly reduces the quantity of key material that must be stored. Furthermore, when a single key is used repeatedly, its value is greatly increased and it becomes a significantly more likely target for compromise.

Simultaneously, long-term keys must be exposed more frequently, in order to facilitate decryption (and in some schemes, encryption as well). When using symmetric-key (secret key) encryption algorithms, a single key is used to both

encrypt and decrypt documents. This allows for potential exposure of keys at encryption time and upon later decryption.

**Key Loss (Destruction):** Loss or destruction of encryption keys renders stored data unrecoverable. Therefore, in any storage system where keys are needed to decrypt data, the system must include a separate mechanism designed to "back up" keys themselves (e.g., storing keys in separate physical locations, etc.) However, each key replica represents a single point of failure, and must be protected. Furthermore, stewardship of keys by multiple parties substantially increases the risk of insider attack.

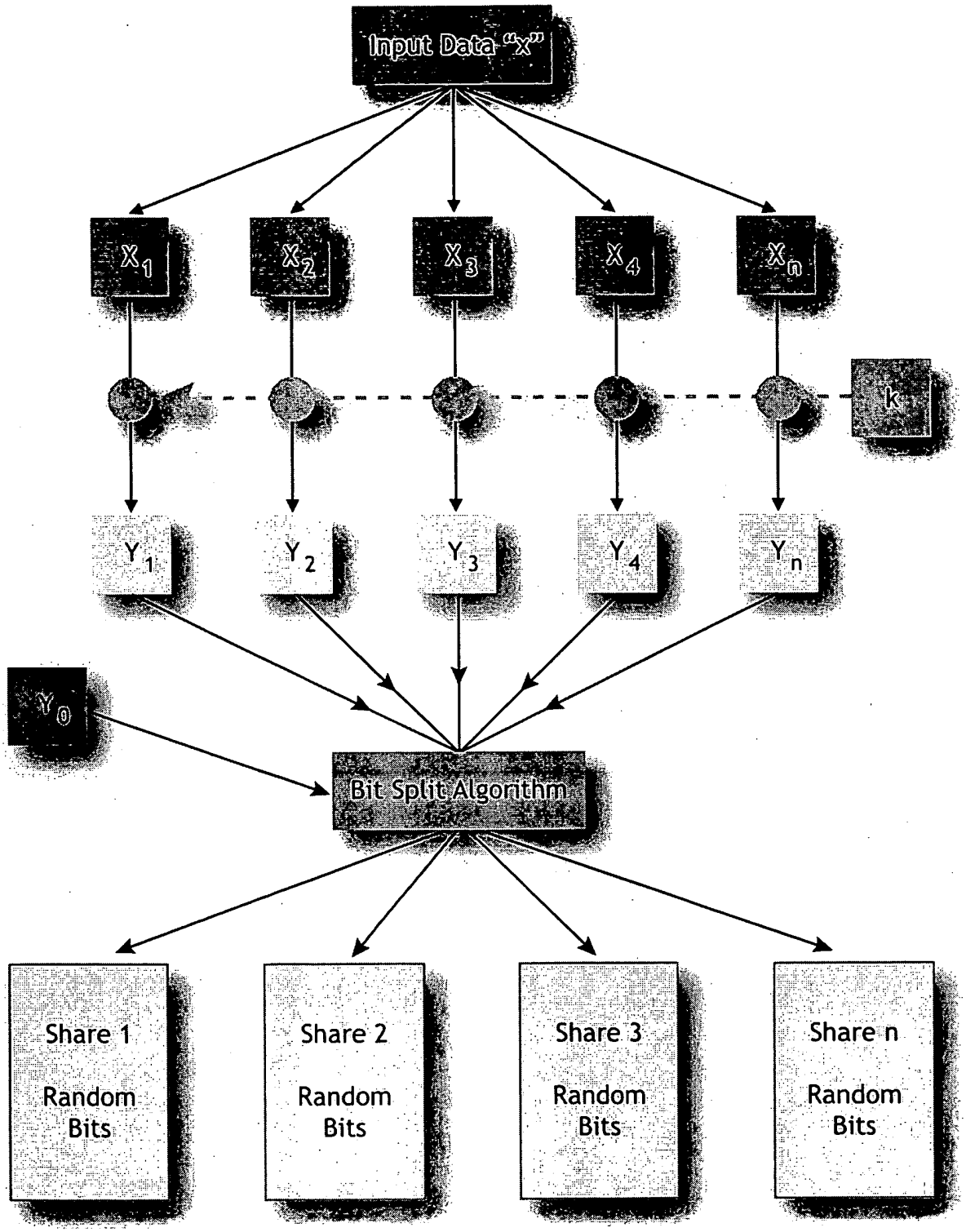
## SOLUTION : SECUREPARSER

The SecureParser product is a complete storage security solution, which offers strong encryption and key management, as well as an innovative approach to "splitting data". The SecureParser system encrypts stored data using the Advanced Encryption Standard (AES), which is sufficient to protect data against even the most resourceful attackers. In addition, the SecureParser provides an efficient and secure "randomized bit level data splitting" approach to securing information, which is more robust and economical to implement and can replace or complement traditional encryption.

**Data Splitting – Parsing:** The data-splitting approach of SecureParser addresses many of the concerns with traditional encryption by providing cryptographic security without complex key management. The SecureParser's data splitting capability works by dividing data into multiple "shares", which are stored in separate locations and or under separate administrative control. The security of data splitting is based on the difficulty of reconstructing the original data without obtaining a pre-specified requisite number of shares. This guarantees that any party who obtains one (or fewer than a pre-specified number) of these shares will learn nothing about the underlying data. Moreover, the SecureParser data splitting approach maps to RAID drives or other high-performance parallel storage systems, and offers many of the same efficiency guarantees.

The following diagram (Figure 1) illustrates the two-step process used by the SecureParser to secure and split data.

**Figure 1:** SecureParser – Functional Flow


Input Data "x"
X1
X2
X3
X4
Xn
k
Y1
Y2
Y3
Y4
Yn
Y0
Bit Split Algorithm
Share 1
Random Bits
Share 2
Random Bits
Share 3
Random Bits
Share n
Random Bits

1. Split input data (x) into "n" number of blocks of preconfigured sizes 128, 192 or 256 bits each in length. The resulting blocks are designated in the diagram as $x_1$ through $x_n$.

2. Generate a random value "k" to be used as the transform key. The random number can be generated from hardware if necessary.

3. Transform each "x" block into a corresponding "y" block using the key value "k".

4. Compute a new y value "$y_0$" to represent the XOR of all available "y" blocks and the key value "k": $y_0 = y_1$ XOR $y_2$ XOR $y_3$ ... $y_n$ XOR k.

5. Discard key value "k".

6. If a session key needs to be held, any y value, including $y_0$, may be optionally held as a session key or encrypted using a public key.

7. Resulting y values including $y_0$ are distributed randomly at the bit level to shares adding fault tolerance as specified.

*To restore the information, all available bits from the shares representing the original "y" blocks must be presented*

As an example, the SecureParser might "split" a multi-gigabyte customer database into four separate shares (each containing approximately ¼ of the total material), such that *all four shares* must be recovered in order to recover the initial file. At the same time, the SecureParser approach provides cryptographic guarantees that no 3 of the shares will reveal *any* useful information about the original data. Similarly, SecureParser might split stored customer data across several file servers which separate administrative controls and authorization mechanisms.

One obvious application for this technique is to protect sensitive data when it is "at rest", as in the case of backup data. In this example, sensitive data might be divided across several backup tapes which could then be stored in separate secure locations. However, SecureParser can also split live data, by dividing it across multiple drives, possibly stored at different locations throughout a network. In either of these examples, an attacker who steals shares, gains no information about the underlying data.

## Advantages

**Multiple Security Levels:** Data splitting is robust enough to be deployed as the sole layer of security in a storage application, and offers strong security guarantees through the use of well-studied cryptographic techniques. When multiple levels of security are desired, the SecureParser's integrated AES

encryption and built-in key management can also be used in combination with data splitting. This combination produces a result that is substantially more resilient to attack than traditional encryption approaches used alone, and provides substantial performance benefits and additional features such as secure deletion.

**Significant Advantage:** Data splitting provides an easily-deployed alternative to complex key management, one that can be understood by all players in a security policy. This is particularly relevant, for applications keys and key management represent a significant obstacle to the deployment of any secure storage. Once split, sensitive data is unrecoverable to an attacker, provided that participants can prevent the loss of some (user-specified) number of shares. This process provides a level of security identical to traditional encryption techniques. Furthermore, SecureParser offers a protection not offered by other encryption approaches, which is a form of "security through bulk", where an attacker must recover most of a potentially very large backup tape in order to obtain useful data. This prevents insiders from memorizing and stealing secrets, as might be possible with decryption keys.

**Cryptographic Security:** The security guarantees of the SecureParser system are equivalent to those provided by traditional encryption approaches. This is due to the fact that the data splitting component of SecureParser is itself built from an federally-certified encryption algorithm (AES), using well-studied cryptographic techniques. In a cryptographic sense, SecureParser can be thought of as an encryption system where the shares themselves combine mathematically to form a key. While data splitting and traditional key-based encryption are different techniques, the constructions used by SecureParser offer the same confidentiality guarantees as AES encryption provided that an attacker does not possess the correct number of shares, i.e., parsed data resists recovery even by an attacker with enormous resources. Moreover, the SecureParser algorithm is highly efficient, and operates at speeds similar to or greater than other secure encryption modes. On some hardware, parsing throughput may be substantially greater, due to the parallelizable algorithms used.

**Faster – Elapsed Time Decreased:** SecureParser data splitting maps perfectly onto parallel-storage solutions such as RAID drives and distributed SAN/NAS stores. By splitting data into shares through a parallelizable process, SecureParser can substantially improve throughput while maintaining security. This allows storage systems to operate at higher capacity by writing to multiple drives, rather than being bound by disk performance. Similarly, SecureParser can substantially decrease backup time by allowing operators to backup onto several tapes simultaneously. Consequently, a large backup job can be performed in a fraction of the time required to write data one tape at a time.

**Physical Separation - Data Storage and Transport Application:** In practice, SecureParser allows for flexible storage policies whereby critical data "at rest" is split onto separate media (e.g., tapes or hard drives), which may then be stored in disparate locations. To prevent loss of sensitive data, it is only necessary to ensure that an attacker is unable to recover a sufficient set of shares to recover the original data. By distributing shares sensibly, an organization can ensure data confidentiality unless an attacker is able to compromise several separate storage locations (or, alternatively, obtain the collusion of a large number of insiders). When data is transported to an offsite location, split data may be transported via independent channels or routes (electronic or physical). An attacker looking to compromise data en route to another location succeeds only If he can compromise many of these transport channels.

**Resilience to Share Loss / Destruction:** Using traditional encryption approaches, data recovery depends on the availability of decryption keys. In that approach, the loss or destruction of a decryption key will prevent any future recovery of an encrypted backup tape, rendering it useless. Protecting against key loss in traditional encryption scenarios requires complex key management techniques, which may not be appropriate for many deployments.

SecureParser includes built-in disaster recovery features to account for the possibility that shares will be lost or destroyed. To allow for data recovery in the event of missing shares, SecureParser allows for the creation of redundant data shares, also called "*m* of *n*" data splits. This technique produces *n* individual shares, but only requires some subset (*m*) of those shares in order to achieve data recovery. This provides a level of redundancy in the event that some shares are not available during the re-construction process. In an example scenario, critical data might be split into four separate shares, which are stored separately. In the event that any one share is damaged or destroyed, the remaining three tapes can be recombined to recover the original data. This "3 of 4" split is one of many possible parameterizations of the parser. In addition to its security properties, this approach can be used to add simple redundancy, as in the case of RAID drives or striped backup tapes.

**Policies & Procedures**: The "*m* of *n*"-split can also be used to create flexible security policies. For example, an operator might divide data into *n* shares for storage by distinct individuals or departments. Based on the settings of SecureParser, any quorum of *m* individuals can combine their shares in order to re-construct data. This addresses a recovery scenario where only certain users are available, but also prevents insider attack by some smaller number of users.

**Mandatory Share - Advanced Options and Secure Deletion:** To provide additional flexibility, SecureParser can also provide for options such as "mandatory shares": shares which are *always* required in order to reconstruct a file. Depending on policy, storage devices may produce an arbitrary number of such shares, which may optionally be sized small enough to fit on portable

devices. These shares serve as a backstop—in a sense, a "master key"—without which the remaining shares cannot be re-combined. This has useful implications for tape re-purposing and destruction. Once a mandatory share has been destroyed, the entire underlying document is effectively destroyed (the remaining shares are will not allow for data recovery). This allows for rapid destruction of data once it is no longer required for storage, and for re-purposing of the remaining tapes or storage devices.

**Efficiency:** The SecureParser algorithms process data at rates similar or greater to other (secure) encryption schemes. SecureParser's AES-based algorithm is optimal, i.e. operates at the same rate as other secure AES-encryption approaches. (Although the SecureParser can be adapted to use other ciphers, AES is recommended due to its security and efficiency.) The SecureParser algorithm offers a set of internal settings, which adjust parameters such as the AES cipher's key length (128/192/256-bit). These decisions can enhance the security level of parsed shares, at some cost to performance.

**Throughput Improvement – MultiPathing:** A primary advantage of SecureParser is the efficiency of the parsing approach. Data "splits" can be performed in parallel by taking advantage of parallel-processing features of the host implementation. Moreover, the encryption schemes used by SecureParser are explicitly designed to take advantage of advanced hardware features such as multi-processing and dedicated cryptographic hardware, in order to multiply data throughput. SecureParser algorithms use a non-sequential encryption mode (AES-CTR) so as to allow parallel processing of data, which allows for many times speedup in the event that these features are available in the underlying hardware.

**Integrity of Data / Authentication:** A concern raised in the referenced document is the possibility of data tampering, e.g., the placement of a Trojan Horse program onto a stored backup. Such tampering can offer a back-channel by which malicious code or data can be inserted into a secure system. SecureParser offers built-in integrity and authentication for shares based on Message Authentication Codes and Digital Signatures (e.g., HMAC-SHA1 and RSA-PSS) . This approach provides cryptographic guarantees against data modification by malicious actors.

## SUMMARY

Given the high costs of data loss, card associations, processors and financial institutions have no choice but to implement secure storage solutions to protect customer data. SecureParser is a complete solution for data protection, and provides a full range of encryption capabilities, including key management, as well as advanced techniques for data-splitting and secure deletion not provided by any other data protection and storage solution. Hence, the secure Parser provides confidentiality, integrity of protection, ease of disaster recovery, along with the ease of implementing a security policy.

## INTRODUCTION

The SecureParser is a unique cross-platform software module suite that not only incorporates NSA Type 1 cryptographic primitives[1] providing an alternative approach to data security, but also has the ability to provide fault tolerance, increased access speed and physical separation of data. This approach eliminates many of the problems typically associated with security solutions including key management and single points of failure.

The cryptographic data-splitting approach of the SecureParser operates by randomly dividing the input data at the bit level into multiple "shares". These shares may be stored in separate locations or transmitted across different channels. Any party who obtains one (or fewer than a pre-specified number) of these shares will be unable to recover the original data. There are several methods that the SecureParser employs for securing and splitting data. One of these methods is discussed in detail in this paper.

The SecureParser allows for additional information to be written to each share, thereby creating a fault tolerant environment that can sustain the loss of one or more shares. For example, the output may be configured to allow for any 3 of 4 shares to recreate the original information. Each share is authenticated during creation and has built-in integrity checks to protect against intentional and unintentional damage.

To provide additional flexibility, the SecureParser also provides for "mandatory shares" which will *always* be required in order to reconstruct the original data. These shares serve as a backstop – in a sense, a "master key" – without which no set of shares can be used to reconstruct the data. Once a mandatory share has been destroyed, the entire underlying data is effectively destroyed. This allows for rapid destruction of data once it is no longer required for storage and can allow for re-purposing of storage assets.

The SecureParser approach simplifies key management and eliminates the need for complex key storage systems. For those installations where keyed encryption type systems are mandated, the SecureParser can be configured to pass a session key back to be stored in a key store.

The security guarantees of the SecureParser system are similar to those provided by traditional encryption approaches. This is due to the fact that the SecureParser is itself a form of un-keyed encryption and uses an NSA Type 1 encryption algorithm (AES)[2] as its core security component. In an alternative sense, the SecureParser can be considered a keyed encryption algorithm where the shares themselves form the key. The construct used by the SecureParser guarantees that the "parsed" output possesses the security properties of the encryption algorithm, which prevents recovery even by an attacker with enormous resources. With the additional attributes of physical separation and fault tolerance, a data security level beyond encryption is now available.

[1] The AES algorithm was certified in 2003 by the NSA for use in Type 1/Type 2 cryptographic modules and by NIST under FIPS for use on Type 3 modules (CNSS Policy No. 15, Fact Sheet No. 1: http://csrc.nist.gov/cryptval/CNSS15FS.pdf, National Information Assurance Glossary: http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf). Certification of an encryption algorithm for classified use does not imply certification for a specific module.

[2] Ibid.

## A DIFFERENT APPROACH TO INFORMATION ASSURANCE

"Traditional" encryption encompasses symmetric and public-key encryption operations, which make use of encryption keys to encipher/decipher data. The practical use of these techniques is complicated primarily by the need to securely perform key storage and management. This necessitates an additional set of procedures for generating, storing, and auditing the use of encryption/decryption keys. Fundamentally, an encryption based approach is no more secure than the procedures applied to the encryption keys. Should keys become compromised before, during or subsequent to a data loss incident, encryption offers no protection for the compromised data.

The SecureParser has advantages in performance over other security solutions. The encryption operations are efficient and operate on small blocks of data. The algorithm is explicitly designed to take advantage of advanced hardware features such as multi-processing and dedicated cryptographic hardware in order to multiply data throughput. A parallelizable encryption mode (AES-CTR) is used so as to allow for parallel processing of data, which allows for significant speed enhancement if those features are available in the underlying hardware. Also, since the output of the SecureParser is multiple shares, each smaller than the original data, multiple channels to storage arrays or communication channels allow for multi-threaded I/O, enhancing speed of access during read and write operations.

In addition, through the use of workgroup keys, the SecureParser can secure information that can only be retrieved by members of a designated group. This can be used in support of Multilevel Security (MLS) solutions for government or hierarchical access for corporate users. In addition, where mandated, the data may be pre-encrypted or any individual share or block of data may be post-encrypted using any encryption algorithm.

The SecureParser is parameter based, allowing for configuration of parameters such as the number of output shares, number of shares required to recombine the data, AES key length (128/192/256 bit) and designation of mandatory shares. The SecureParser can integrate easily with PKI, securing any individual key value or share with a public key, and allowing only the owner of the private key the ability to restore the data.

The following diagram (Figure 1) outlines the functional flow of the SecureParser process in more detail.

1. Split input data (x) into "n" number of blocks of preconfigured sizes 128, 192 or 256 bits each in length. The resulting blocks are designated in the diagram as $x_1$ through $x_n$.

2. Generate a random value "k" to be used as the transform key. The random number can be generated from hardware if necessary.

3. Transform each "x" block into a corresponding "y" block using the key value "k".

4. Compute a new y value "$y_o$" to represent the XOR of all available "y" blocks and the key value "k": $y_o = y_1$ XOR $y_2$ XOR $y_3$ ... $y_n$ XOR k.

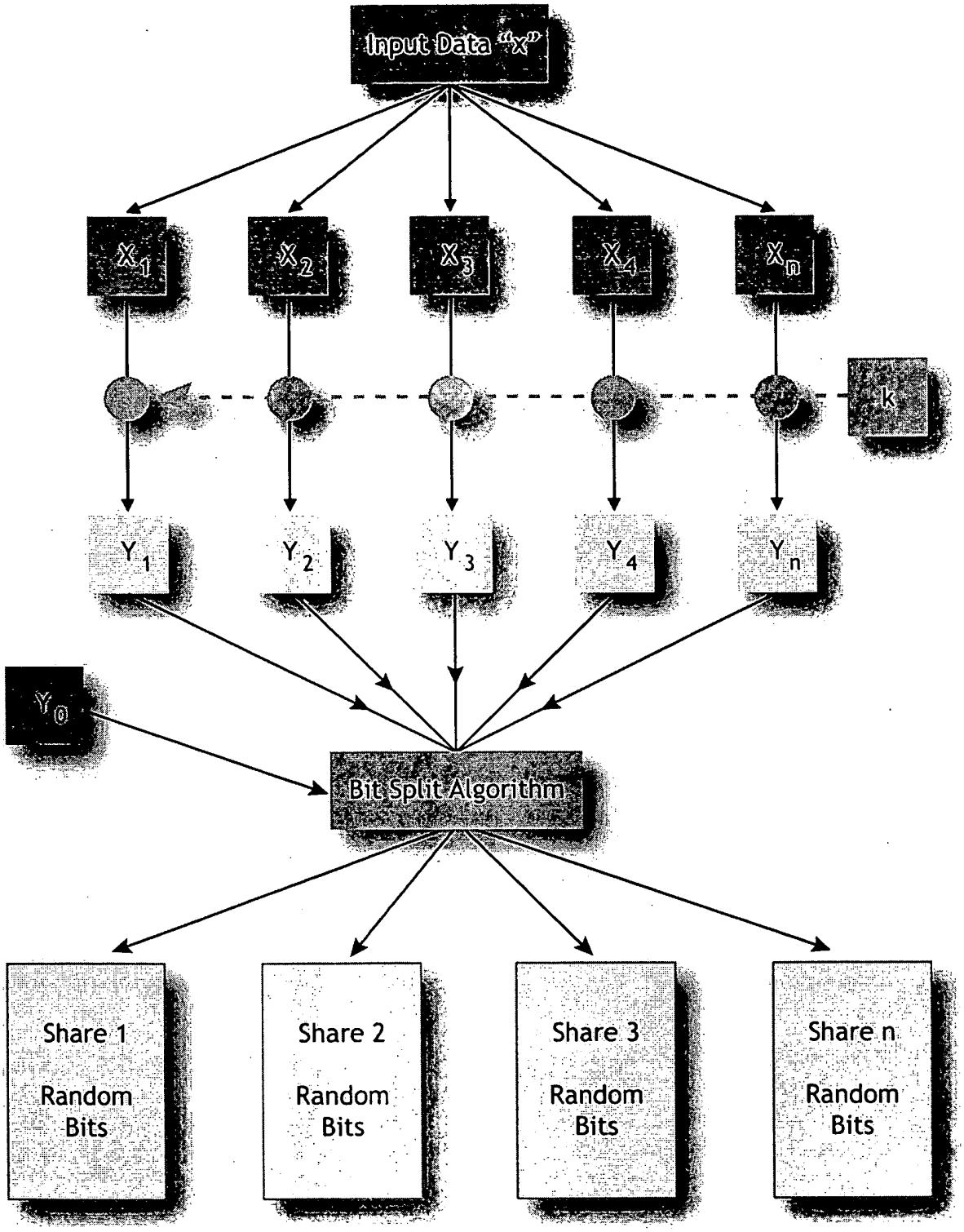
5. Discard key value "k".

6. If a session key needs to be held, any y value, including $y_0$, may be optionally held as a session key or encrypted using a public key.

7. Resulting y values including $y_0$ are distributed randomly at the bit level to shares adding fault tolerance as specified.

*To restore the information, all available bits from the shares representing the original "y" blocks must be presented*

**Figure 1:** SecureParser – Functional Flow


Input Data "x"
X1
X2
X3
X4
Xn
k
Y1
Y2
Y3
Y4
Yn
Y0
Bit Split Algorithm
Share 1
Random
Bits
Share 2
Random
Bits
Share 3
Random
Bits
Share n
Random
Bits

## HOW IS THE SECUREPARSER INTEGRATED TO PROTECT DATA AT REST OR IN MOTION?

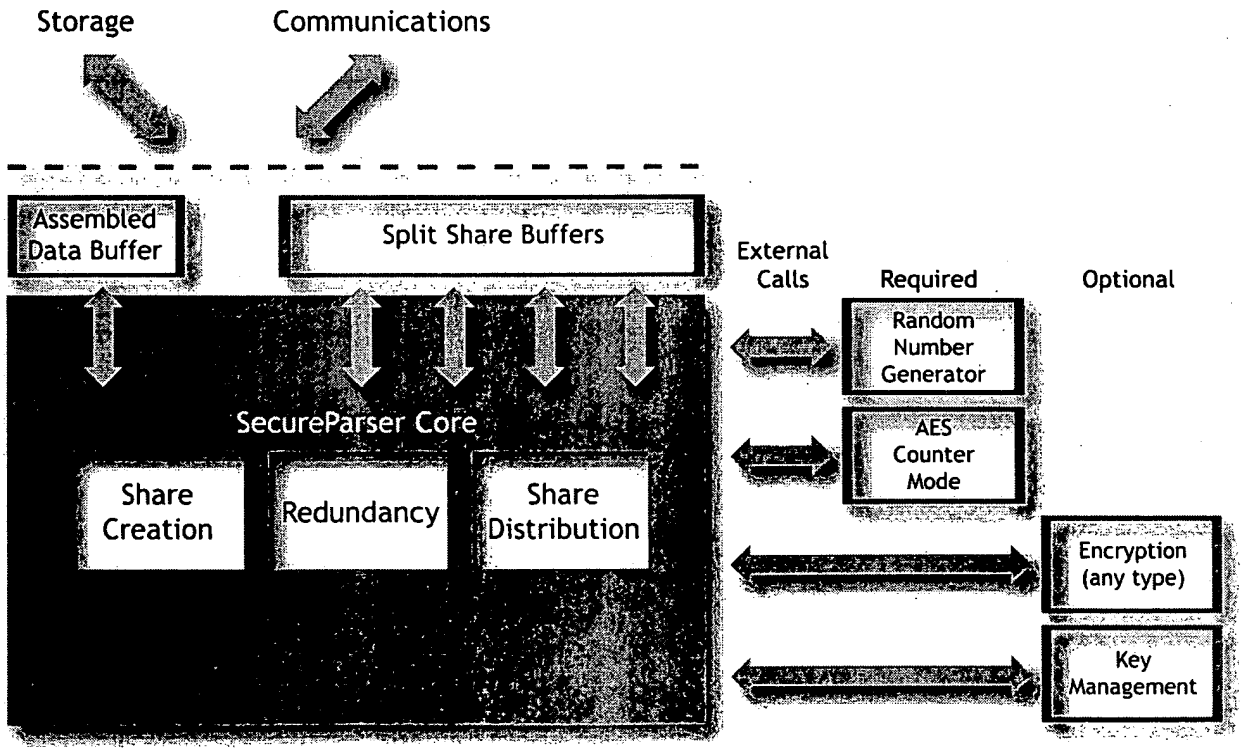
The SecureParser Core (Figure 2) isolates the functions of data splitting (share creation, redundancy, and distribution to share buffers) into a callable module. Data is placed in the Assembled Data Buffer to be split and the SecureParser is invoked. The resulting shares are placed into the Split Shares Buffers to be securely stored or transmitted by the host application.

All cryptographic functions are external calls for the SecureParser. This allows for the custom integration of various cryptographic libraries, giving flexibility for "bolting on" other encryption modules, key management systems, random number generators, etc. as needed.

External functions called by the SecureParser are:

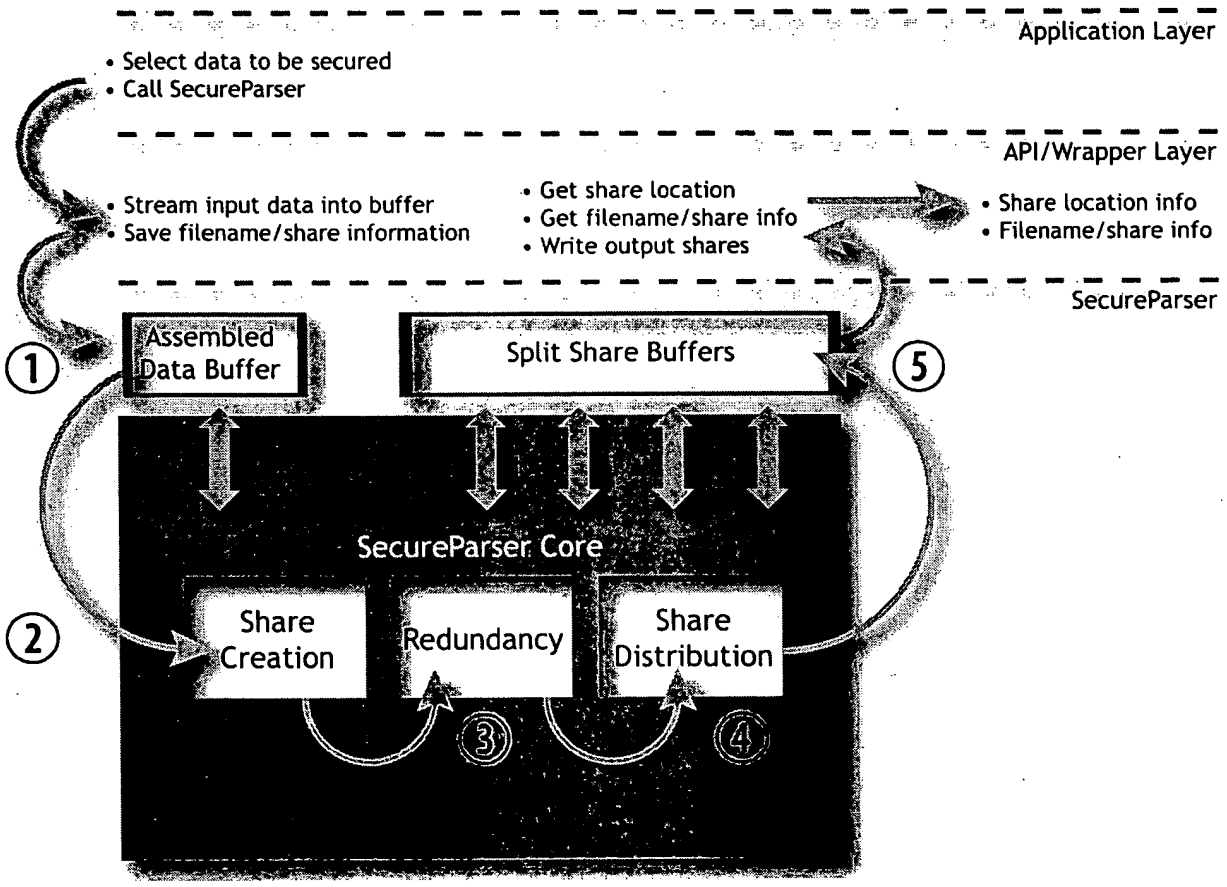
- Random Number Generator – For each split operation, a unique random value is generated to be used as the basis for the split. Random numbers can be generated in a number of ways, via software program or hardware interface.
- AES Counter Mode Module – This module secures each block of data prior to the distribution to the shares.
- Encryption Algorithm – In some circumstances the data must be pre-encrypted. If this is the case, the SecureParser can invoke external encryption to process the data prior to the split. Alternatively, shares may be encrypted post-splitting.
- Key Management System – In some cases, external key management will be mandated. If so, the SecureParser will pass a key value to an external system for secure storage.

**Figure 2:** SecureParser Core



Integration to the SecureParser is accomplished through a well defined Application Programming Interface (API). Since the SecureParser is designed to be integrated to any application or infrastructure component which stores or transmits data, there are no references within the SecureParser to where the data is stored or transmitted. Data is placed into the inbound SecureParser buffer to be split, and the resulting split shares are picked up by the host application. In many cases, an API integration layer is an effective way to reduce the amount of coding needed for a particular environment. The API integration layer defines the location of the resulting shares, such as paths to send the information or storage device to store the shares. The API integration layer also defines the naming convention of the resulting data, or parameters to store with the information such as row, column and table ID in the case of a database application or the original filename in the case of unstructured data such as documents, images, maps, multimedia, or streaming data. The diagram below (Figure 3) shows the API integration layer. Overall, this API layer handles streaming of the data to the SecureParser and secure storage or communications of the resulting shares.

**Figure 3:** SecureParser Integration



## WHAT MAKES THIS TECHNOLOGY DIFFERENT FROM TYPICAL ENCRYPTION METHODS?

There are several key advantages of the SecureParser over typical encryption methods:

1. The SecureParser offers keyless security, eliminating the complexity of generation, storage and long term management of encryption keys.
2. Physical separation of data allows for flexible data storage and transmission policies which are unable to be achieved using existing encryption.
3. The fault tolerant mode of the SecureParser can allow for faster access to data, disaster recovery using the remaining shares, and can eliminate the need for data backup.
4. The system is easily deployed and can be understood by all players in a security policy.
5. The SecureParser offers improved speed of encryption through a parallelizable encryption mode, and faster I/O operations through multi-threading of read and write operations using smaller shares of data than the original.

## WHY IS IT SECURE?

The SecureParser employs strong NSA Type 1 cryptographic primitives as its core security component. The construction used by the SecureParser guarantees that the parsed output possesses the security properties of the encryption algorithm. As a construct to improve security over typical encryption solutions, the SecureParser uses randomized bit level physical separation of all data into discrete shares, eliminating single points of failure.

## HOW IS ACCESS TO THE DATA MONITORED?

A common concern raised is the possibility of data tampering such as the placement of a malicious program. Such tampering can offer a back-channel by which malicious code or data can be inserted into a secure system. The SecureParser offers built-in integrity and authentication for shares based on Message Authentication Codes and Digital Signatures. This approach provides cryptographic guarantees against data modification.

The SecureParser also has an available audit module that logs each time a user or program requests that data be recombined. This module ensures that a record of authorized (and unauthorized) attempts at data access has been recorded. For additional security, the resulting audit log may be parsed and distributed to multiple locations to prevent tampering with the information.

## WHAT ABOUT DATA RECOVERY?

The SecureParser was designed with the ability to be deployed without the need for backup. Offsite storage of backups, redundant systems, mirrors and replicated data create an additional security risk by exposing the entirety of the data in other locations and formats. A built in fault tolerance option in the SecureParser allows for additional information to be written to each resulting share, allowing for the recreation of the original information from less than the total number of shares. For example, using a 2 of 4 distribution model, only 2 shares would be needed to recreate the original data. This allows for the recreation of a lost or damaged share by any 2 of the remaining 3 shares and may eliminate the need for backup or replication of data.

Integrity of the individual shares is assured by the use of an authentication value embedded in the individual shares. This integrity value ensures that the share has the required information to recreate the original data and has not been tampered with or damaged. Use of a utility program to verify integrity will proactively validate the information and send an alert if a share has been damaged in any manner.

## KEY MANAGEMENT

The SecureParser method allows for keyless information security, eliminating the complexities of current keyed encryption solutions. The SecureParser is also compatible with existing keyed systems by optionally withholding a key value to be stored if desired and /or required.

Discussed below are some issues associated with traditional key management:

**Key Generation:** When unique encryption keys are generated for each operation, each new key must be stored. This can be done using a local key-store. However, in this scenario, key material may be vulnerable to compromise along with the data. This concern can be mitigated through the use of tamper-resistant hardware, key encryption, offsite key storage, etc. However, many of these approaches serve only to shift the problem of key storage to another potential point of failure.

**Key Exposure:** When using symmetric-key encryption algorithms, a single key is used to both encrypt and decrypt data. This allows for potential exposure of keys at encryption and at a later decryption point. The use of public key cryptography can mitigate this problem by providing a separate non-secret key for use specifically when encrypting data (this key cannot be used to decrypt the data). However, data recovery still requires access to the secret decryption key, which must be conveyed from a secure location and used by a responsible party.

**Key Loss (Destruction):** Loss or destruction of encryption keys renders stored data unrecoverable. Therefore, in any approach to data backup where keys are required for recovery, the system should include a separate mechanism designed to "back-up" keys themselves (e.g., storing keys in a separate physical location). However, each key replica represents a single point of failure and must be protected. Furthermore, stewardship of keys by multiple parties substantially increases the risk of insider attack. This can be mitigated through key-splitting approaches where multiple "split" keys must be re-combined to recover an encryption key. The SecureParser provides this functionality using a more flexible security mechanism, similar to the two man rule for missile launching or a kill switch mechanism.

## SUMMARY

The SecureParser offers unmatched flexibility in information security and access. At the core, strong NSA Type 1 cryptographic primitives[3] protect the data in a keyless encryption offering, eliminating the single point of failure in other data security solutions. Physical separation allows for enhanced protection of data, survivability and business continuity, as well as potential performance increases in multi-threaded read/write operations. The SecureParser's "m of n" fault tolerance mode offers improved access speed, assembling the data when the first "m" shares are retrieved. This mode also allows for reconstruction of any share from the remaining shares in case of an intentional or unintentional failure. The mandatory share option allows for repurposing of storage media by the simple destruction of a single block of data.

The SecureParser is small, efficient, and portable and may be run in any number of environments, from servers to desktops, cell phones to PDAs, and dedicated appliances to smartcards.

# Secured Storage


## ABSTRACT

Securing storage data is a manifold problem with requirements in three dimensions: data security, data integrity, and the safety of data. Meeting the requirements for one dimension often means compromising another. SecureParser™ is a software technology which addresses all three dimensions of secure storage without compromising any. In this paper, we describe the SecureParser™ technology and discuss how it addresses the three dimensions of secured storage: security, integrity, and safety.


## 1. INTRODUCTION

The SecureParser™ from Security First Corp. is a unique, patent pending, cross-platform software module suite that not only incorporates NSA cryptographic primitives providing an alternative approach to data security, but also has the ability to provide fault tolerance, increased access speed and physical separation of data. This approach eliminates many of the problems typically associated with security solutions including key management and single points of failure and data unavailability.

The cryptographic data-splitting approach of the SecureParser™ operates by randomly dividing the input data at the bit level into multiple "shares". These shares may be stored in separate locations or transmitted across different communication channels. Any party who obtains or holds one (or less than a pre-specified number) of these shares will be unable to recover the original data. There are several methods that the SecureParser™ employs for securing and splitting data. One of these methods is discussed in detail in this paper.

The SecureParser™ allows for additional information to be written to each share, thereby creating a fault tolerant environment that can sustain the loss of one or more shares. For example, the output may be configured to allow for any 3 of 4 shares to recreate the original information. Each share is authenticated during creation and has built-in integrity checks to protect against intentional and unintentional damage.

The SecureParser™ approach simplifies key management and eliminates the need for complex key storage systems. For those installations where keyed encryption type systems are mandated, the SecureParser™ can be configured to pass a session key back to be stored in a key store. The security guarantees of the SecureParser™ system are similar to those provided by traditional encryption approaches. This is due to the fact that the SecureParser™ is itself a form of un-keyed encryption and uses an NSA registered encryption algorithm (AES) as its core security component. In an alternative sense, the SecureParser™ can be considered a "keyed" encryption algorithm where the shares themselves form the key. The construct used by the SecureParser™ guarantees that the "parsed" output possesses the security properties of the encryption algorithm, which prevents recovery even by an attacker with enormous resources. With the additional attributes of physical separation and fault tolerance, a data security level distinct from traditional, serial encryption is now available.

## 2. PROBLEM STATEMENT

To ensure the quality of data, information management systems must guarantee the safety, integrity, and security of that data.

The term "data safety" in the context of this paper refers to the requirement that data can not be lost through accident or attack. Typically, rigorous backup procedures are implemented to periodically copy all data to offsite, geographically dispersed, locations. Of course, there are windows in time when data is at risk, specifically between when a file is modified and when it is backed up. Mirroring technologies help close the timing holes, but at a high cost. In general, ensuring data safety in traditional ways incurs high costs in manpower and equipment.

"Data integrity" means that data can not be changed, either through accident or by design. Enterprise-class IT equipment protects against data corruption with hardware mechanisms such as parity and error-correcting codes. More insidious, however, are malicious modifications that go undetected until they have caused system crashes or erroneous decisions. Check summing and other techniques such as digital watermarking may be employed by software to guarantee the integrity of data files, but those techniques are processor intensive and are often not applied consistently across and enterprise's digital assets.

Finally, "data security" means that data can not be read or copied by unauthorized agents. Maintaining safety and security of data often means working at cross purposes. To be safe, data should be duplicated as many times as necessary to ensure a copy will survive the worst case disaster. This could very well mean multiple, remote backups and/or replication of live data to disaster recovery sites. On the flip side, to be secure, data should exist in only one place, since the more data is replicated, the more likely it is to be stolen.

SecureParser™ and the methods it may be employed, as described herein, can maximize all three aspects of data safety, integrity, and security in a coherent manner.

## 2.1 Solving the Data Security Requirement

"Traditional" encryption employ private or public-key encryption operations, which make use of encryption keys to encipher/decipher data in a serial or streamed fashion. The practical use of these techniques is complicated primarily by the need to securely perform key storage and key management. This necessitates an additional set of procedures for generating, storing, and auditing the use of encryption/decryption keys. Fundamentally, an encryption based approach is no more secure than the procedures applied to the encryption keys. Should keys become compromised before, during or subsequent to a data loss incident, encryption offers no protection for the compromised data.

The SecureParser[tm] has advantages in performance over other security solutions. The encryption operations are efficient and operate on small blocks of data. The algorithm is explicitly designed to take advantage of advanced hardware features such as multi-processing and dedicated cryptographic hardware in order to multiply data throughput performance. A 'parallelizable' encryption mode (AES-CTR) is used so as to allow for parallel processing of data, which allows for significant speed enhancement if those features are available in the underlying hardware. Also, since the output of the SecureParser[tm] is multiple shares, each smaller than the original data, multiple channels to storage arrays or communication channels allow for multi-threaded I/O, enhancing speed of access during read and write operations.

The SecureParser[tm] is parameter based, allowing for configuration of parameters such as the number of output shares, number of shares required to recombine the data, AES key length (128/192/256 bit) and designation of mandatory shares. The SecureParser[tm] can integrate easily with PKI, securing any individual key value or share with a public key, and allowing only the owner of the private key the ability to restore the data.

The following diagram (Figure 1) outlines the functional flow of the SecureParser[tm] process in more detail.

1. Split input data (x) into "n" number of blocks of preconfigured sizes 128, 192 or 256 bits each in length. The resulting blocks are designated in the diagram as $x_1$ through $x_n$.
2. Generate a random value "k" to be used as the transform key. The random number can be generated from hardware if necessary.
3. Transform each "x" block into a corresponding "y" block using the key value "k".
4. Compute a new y value "$y_0$" to represent the XOR of all available "y" blocks and the key value "k": $y_0 = y_1$ XOR $y_2$ XOR $y_3$ ... $y_n$ XOR k.
5. Discard key value "k".
6. If a session key needs to be held, any y value, including $y_0$, may be optionally held as a session key or encrypted using a public key.
7. Resulting y values including $y_0$ are distributed randomly at the bit level to shares adding fault tolerance as specified.

To restore the information, all available bits from the shares representing the original "y" blocks must be presented

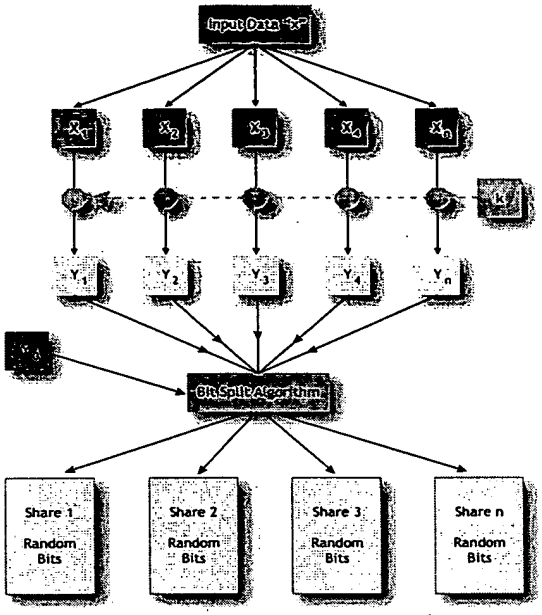



**Figure 1. SecureParser[tm] Functional Flow**

# 3. SECUREPARSER[TM] TECHNOLOGY

## 3.1 Description of Method

The SecureParser[tm] Core (Figure 2) isolates the functions of data splitting (share creation, redundancy, and distribution to share buffers) into a callable module. Data is placed in the Assembled Data Buffer to be split and the SecureParser[tm] is invoked. The resulting shares are placed into the Split Shares Buffers to be securely stored or transmitted by the host application.

All cryptographic functions are external calls for the SecureParser[tm]. This allows for the custom integration of various cryptographic libraries, giving flexibility for "bolting on" other encryption modules, key management systems, random number generators, etc. as needed.

External functions called by the SecureParser[tm] are:

- Random Number Generator – For each split operation, a unique random value is generated to be used as the basis for the split. Random numbers can be generated in a number of ways, via software program or hardware interface.
- AES Counter Mode Module – This module secures each block of data prior to the distribution to the shares.
- Encryption Algorithm – In some circumstances the data must be pre-encrypted. If this is the case, the SecureParser[tm] can invoke external encryption to process the data prior to the split. Alternatively, shares may be encrypted post-splitting.

- Key Management System – In some cases, external key management will be mandated. If so, the SecureParser[tm] will pass a key value to an external system for secure storage.

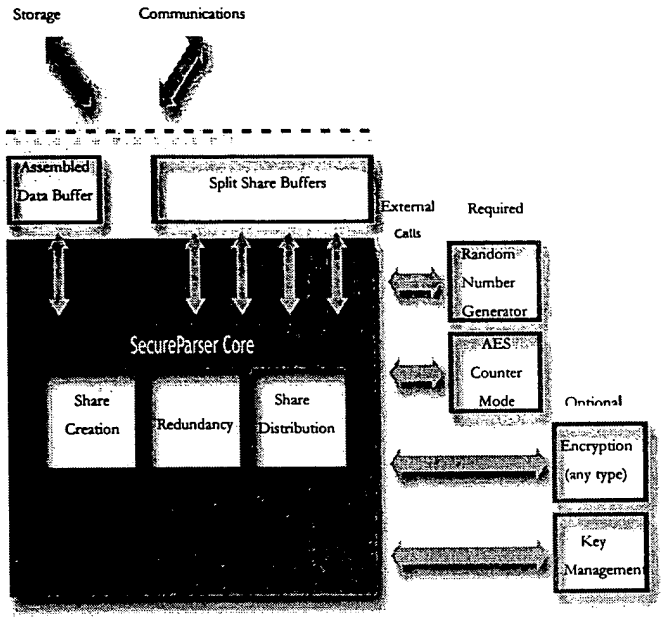



**Figure 2. SecureParser[tm] Core**

Integration to the SecureParser[tm] is accomplished through an Application Programming Interface (API). Since the SecureParser[tm] is designed to be integrated to any application or infrastructure component which stores or transmits data, there are no references within the SecureParser[tm] to where the data is stored or transmitted. Data is placed into the inbound SecureParser[tm] buffer to be split, and the resulting split shares are picked up by the host application. In many cases, an API integration layer is an effective way to reduce the amount of coding needed for a particular environment. The API integration layer defines the location of the resulting shares, such as paths to send the information or storage device to store the shares. The API integration layer also defines the naming convention of the resulting data, or parameters to store with the information such as row, column and table ID in the case of a database application or the original filename in the case of unstructured data such as documents, images, maps, multimedia, or streaming data. The diagram below (Figure 3) shows the API integration layer. Overall, this API layer handles streaming of the data to the SecureParser[tm] and secure storage or communications of the resulting shares.

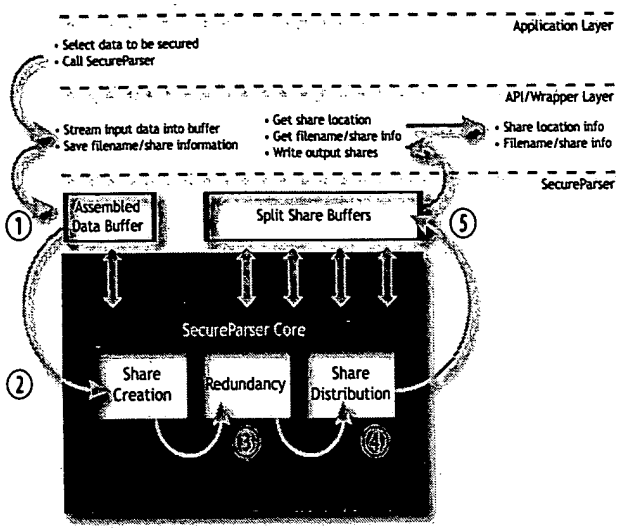



**Figure 3. SecureParser[tm] API Integration Layers**

## 3.2 Integration Points

There are several points within the data communications architecture where the SecureParser[tm] technology can be integrated. These integration points and their pros and cons are shown in Table 1.

**Table 1. SecureParser[tm] Integration Pros and Cons**

| Layer | Pros | Cons |
|---|---|---|
| Application | Custom integration for application differentiation and value-add. (Currently packaged with installation tools.) | Customization required for each application |
| File system | Generic file access;<br>Local and remote files supported;<br>Share/key mapping integrated into directory;<br>Concurrent multiple paths | System software update required |
| Device driver | LUN-based rather than file-based;<br>Concurrent multiple paths;<br>Supports legacy HBAs and fabrics | System software update required |
| HBA | Offloads parsing overhead from host CPU;<br>Legacy fabrics supported | System software update required |
| Switch | Parsing is transparent to hosts;<br>Highest performance | Switch-accessible LUN map required;<br>Specific switch hardware required |

An example of a Parsed File System is outlined below. Because implementation of SecureParser[tm] at the device driver and/or HBA layers requires hardware and/or system software updates to all servers (as does a PFS), these approaches are less practical than

integrating SecureParser™ into the fabric itself by implementing it within the switches.

In the sections below, the Parsed File System and Parsed Fabric options are examined.

## 3.3 Parsed File System

The Parsed File System (PFS) is a distributed file system which incorporates a directory, shared among the file servers, which is itself parsed. The PFS directory is the clients' view of all file assets. The set of files visible to an individual client is controlled by that client's group keys. An example is given in the next section.

PFS features include:

1) The distribution of SecureParser™ shares is hidden from the client. The file server location and the names of the files containing the share data (i.e. the URLs of the shares) are maintained within the directory. The file system client is given access to those shares at file open.
2) In addition to the normal file attributes such as timestamps, size, access privileges, etc., the directory stores:
   a. N-of-M
   b. URLs or some other globally unique identifier (GUID) of the shares
   c. a Key-Id referencing the actual SecureParser™ key which was used to parse the file. The file server verifies at open time that the client has been granted the key referenced by the Key-Id.
   d. Validity check data
3) The directory data is parsed with separate, replaceable group keys to allow keys to be rescinded and replenished as required, for example when personnel changes or suspected security breaches warrant.
4) Open and close operations are audited and any suspected declassifications (movement of data from a higher security level to a lower security level) are sequestered for authorization. The sequestration method is as follows:
   a. As data is read from a file, a signature of the data is calculated every S bytes (S is a tunable parameter).
   b. Signatures, along with the file's group key are stored by the file server until all of the following are true:
      i. the file is closed
      ii. all processes running when the file was opened are terminated
      iii. all processes which were started after the file was opened but while there was data in the clipboard are terminated
   c. Target signatures are generated on writes and compared to read signatures of files with different group keys to detect suspicious cut-and-paste operations.
5) Files parsed with group keys not included in the user's authorized key set are not visible to that user in the directory.
6) Signature generators for different file types (e.g. .TXT, .WRD, .XLS, .JPG) can be added to PFS through a plug-in interface.
7) Group keys can be defined to inherit privilege from other keys. For example:
   a. Unclassified = {UNCLASS}
   b. Secret = {SECRET, UNCLASS}
   c. TS/SCI = {TS_SCI, SECRET, UNCLASS}
   d. Special1 = {SAP1, SECRET, UNCLASS}
   e. Special2 = {SAP2, TS_SCI, SECRET, UNCLASS}
   f. Special3 = {SAP3, SAP1, TS_SCI, SECRET, UNCLASS}

### *3.3.1 PFS Example*

A sample use-case illustrates the PFS behavior.

1) USER1 logs in to the network and is granted {USER1,SAP1,SECRET,UNCLASS} keys.
2) USER1 opens Notepad and types "The quick fox jumped over the lazy brown dog."
3) USER1 saves the file as TEST1.TXT.
4) A dialog box appears asking what SecureParser™ group key should be used: USER1, SAP1, SECRET, or UNCLASS.
5) USER1 chooses SECRET and TEST1.TXT is saved.
6) USER2 logs in and is granted {USER2,UNCLASS} keys.
7) TEST1.TXT is not visible when USER2 opens Explorer.
8) USER3 logs in and is granted {USER3,TS_SCI,SAP1,SECRET,UNCLASS} keys.
9) USER3 sees TEST1.TXT and opens it.
10) USER3 copies "quick fox" to clipboard.
11) USER3 pastes clipboard ("quick fox") to a new file.
12) USER3 saves the new file as TEST2.TXT and picks the UNCLASS group key.
13) A warning dialog box appears with the message "TEST2.TXT will be sequestered for declassification review."
14) USER3 cancels the warning – the new file is not saved.
15) USER3 saves TEST2.TXT using SAP1 key. Since SAP1 includes SECRET, which was TEST1.TXT's key, no warning is issued.
16) USER1 sees and reads TEST2.TXT (since he has the SAP1 key), but USER2 can not.
17) USER3 copies/pastes "brown dog" to a new file and saves the new file as TEST3.TXT using key UNCLASS.
18) The declassification warning appears and USER3 OKs it.
19) ADMIN logs in and is granted all keys.
20) ADMIN receives a message to review TEST3.TXT for possible declassification.
21) ADMIN shows Details>>>
    File TEST3.TXT, written by USER3 using key UNCLASS. It contains "brown dog", which may have been copied from TEST1.TXT (key SECRET).
22) ADMIN OKs the save.
23) USER1, USER2, and USER3 all see and can access TEST3.TXT.

## 3.4 Parsed Fabric

An alternative to the Parsed File System is an approach which pushes the parsing operation into the storage fabric. Rather than parsing individual files, specific logical unit numbers (LUNs), i.e. drives, are parsed.

In the method shown above, the host application and server have no involvement in the parsing process. They simply issue standard operations (Read, Write, etc.) into the fabric. The Parser-enabled Switch is responsible for accessing the Key Manager and a LUN Map, somewhere in the fabric, which contains a mapping from application-visible LUNs to physical locations of the corresponding shares.

By pushing the parsing function into the fabric, all parsing processing is offloaded from the servers. More importantly, no changes are required to application, device driver, or operating system software. In environments where rigorous certification processes are used to qualify all software, being host-transparent is a significant advantage.

Of course, the Parser-enabled Switch must be certified to interoperate with all of the appropriate fabric and storage equipment.

## 4. Differentiators

There are several key advantages of the SecureParser[tm] over typical encryption methods:

- The SecureParser[tm] offers keyless security, eliminating the complexity of generation, storage and long term management of encryption keys.
- Physical separation of data allows for flexible data storage and transmission policies which are unable to be achieved using existing encryption.
- The fault tolerant mode of the SecureParser[tm] can allow for faster access to data, disaster recovery using the remaining shares, and can eliminate the need for data backup.
- The system is easily deployed and can be understood by all players in a security policy.
- The SecureParser[tm] offers improved speed of encryption through a parallelizable encryption mode, and faster I/O operations through multi-threading of read and write operations using smaller shares of data than the original.

### 4.1 Why Is It Secure?

The SecureParser[tm] employs strong NSA registered cryptographic primitives as its core security component. The construction used by the SecureParser[tm] guarantees that the parsed output possesses the security properties of the encryption algorithm. The SecureParser[tm] not only employs strong NSA registered cryptography at its core, but uses bit-level physical separation of all the data as a construct to improve security over typical encryption solutions, eliminating single points of failure.

### 4.2 Data Recovery

The SecureParser[tm] was designed with the ability to be deployed without the need for traditional backup practices. Offsite storage of backups, redundant systems, mirrors and replicated data create an additional security risk by exposing the entirety of the data in other locations and formats. A built in fault tolerance option in the SecureParser[tm] allows for additional information to be written to each resulting share, allowing for the recreation of the original information from less than the total number of shares. For example, using a 2 of 4 distribution model, only 2 shares would be needed to recreate the original data. This allows for the recreation of a lost or damaged share by any 2 of the remaining 3 shares and may eliminate the need for backup or replication of data.

Integrity of the individual shares is assured by the use of an authentication value embedded in the individual shares. This integrity value ensures that the share has the required information to recreate the original data and has not been tampered with or damaged. Use of a 'utility' program module to verify integrity will proactively validate the information and send an alert if a share has been damaged in any manner.

### 4.3 Key Management

The SecureParser[tm] method allows for keyless information security, eliminating the complexities of current keyed encryption solutions. Discussed below are some issues associated with key management.

#### *4.3.1 Key Generation*

When unique encryption keys are generated for each operation, each new key must be stored. This can be done using a local key-store. However, in this scenario, key material may be vulnerable to compromise along with the data. This concern can be mitigated through the use of tamper-resistant hardware, key encryption, offsite key storage, etc. However, many of these approaches serve only to shift the problem of key storage to another potential single point-of-failure.

#### *4.3.2 Key Exposure*

When using symmetric-key encryption algorithms, a single key is used to both encrypt and decrypt data. This allows for potential exposure of keys at encryption and at a later decryption point. The use of public key cryptography can mitigate this problem by providing a separate non-secret key for use specifically when encrypting data (this key cannot be used to decrypt the data). However, data recovery still requires access to the secret decryption key, which must be conveyed from a secure location and used by a responsible party.

#### *4.3.3 Key Loss*

Loss or destruction of encryption keys renders stored data unrecoverable. Therefore, in any approach to data backup where keys are required for recovery, the system should include a separate mechanism designed to "back-up" keys themselves (e.g., storing keys in a separate physical location). However, each key replica represents a single point of failure and must be protected. Furthermore, stewardship of keys by multiple parties substantially increases the risk of insider attack. This can be mitigated through key-splitting approaches where multiple "split" keys must be re-combined to recover an encryption key. The SecureParser[tm] provides this functionality using a more flexible security mechanism, similar to the two man rule for missile launching.

## 5. SUMMARY

The SecureParser[tm] offers unmatched flexibility in information security and access. At the core, strong NSA cryptographic primitives protect the data in a keyless encryption offering, eliminating the single point of failure in other data security solutions. Physical separation allows for enhanced protection of data, survivability and business continuity, as well as potential performance increases in multi-threaded read/write operations. The SecureParser's[tm] "m of n" fault tolerance mode offers improved access speed, assembling the data when the first "m" shares are retrieved. This mode also allows for reconstruction of any share from the remaining shares in case of an intentional or unintentional failure. The mandatory share option allows for

repurposing of storage media by the simple destruction of a single block of data.

The SecureParser[tm] is small, efficient, and portable and may be run in any number of environments, from servers to desktops, cell phones to PDAs, and dedicated appliances to smartcards.

# Table of Contents


**1. INTRODUCTION ..... 3**

1.1. DESCRIPTION ..... 3
1.2. INSTALLATION ..... 3
*1.2.1. Disk Content ..... 3*
*1.2.2. Install ..... 3*

**2. DATA TYPES AND STRUCTURES ..... 4**

2.1. BASE TYPES, STRUCTURES, AND ENUMERATORS ..... 4
2.2. ENUMERATORS ..... 4
*2.2.1. ENC_TYPE ..... 4*
*2.2.2. AUTH_TYPE ..... 5*
*2.2.3. HASH_TYPE ..... 5*
*2.2.4. SPLIT_TYPE ..... 5*
*2.2.5. KEY_TYPE ..... 5*
*2.2.6. ERROR_TYPE ..... 6*
2.3. DATA STRUCTURES ..... 6
*2.3.1. Parser: ..... 6*
*2.3.2. ParserParams: ..... 6*
*2.3.3. KeyId: ..... 7*
*2.3.4. KeyShare: ..... 7*
*2.3.5. Key: ..... 7*
*2.3.6. Share: ..... 8*
*2.3.7. EncContext: ..... 8*
*2.3.8. EncContext_AESCBC ..... 8*
*2.3.9. EncContext_AESCTR ..... 8*
*2.3.10. AuthContext ..... 9*
*2.3.11. AuthContext_RSAPSS ..... 9*
*2.3.12. AuthContext_DSA ..... 9*
*2.3.13. AuthContext_ECSA ..... 9*
*2.3.14. AuthContext_HMACSHA1 ..... 9*
*2.3.15. HashContext ..... 10*
*2.3.16. HashContext_SHA ..... 10*
2.4. DATA FORMAT: ..... 11

**3. FUNCTION LIBRARY REFERENCE ..... 13**

3.1. PARSER_CREATEPARSER ..... 13

3.2. PARSER_DESTROYPARSER ..... 13
3.3. PARSER_CREATEKEY ..... 13
3.4. PARSER_DESTROYKEY ..... 13
3.5. PARSER_CREATESHARE ..... 13
3.6. PARSER_DESTROYSHARE ..... 14
3.7. PARSER_INITPARAMETERS ..... 14
3.8. PARSER_SETPARAMETERS ..... 14
3.9. PARSER_SETFINALIZE ..... 14
3.10. PARSER_GETORIGINALSHARE ..... 14
3.11. PARSER_GETSHARE ..... 15
3.12. PARSER_DOSPLIT ..... 15
3.13. PARSER_DORESTORE ..... 15
3.14. PARSER_GENERATEHEADERS ..... 15
3.15. PARSER_GENERATEFOOTERS ..... 15
3.16. PARSER_RESTOREHEADERS ..... 15
3.17. PARSER_RESTOREFOOTERS ..... 16
3.18. PARSER_GETRESTORESTATUS ..... 16
**4. FUNCTION DETAILS ..... 17**
4.1. PARSER_GENERATEHEADERS ..... 17
4.2. PARSER_DOSPLIT ..... 18
4.3. PARSER_GENERATEFOOTERS ..... 19
4.4. PARSER_RESTOREHEADERS ..... 20
4.5. PARSER_DORESTORE ..... 21
4.6. PARSER_RESTOREFOOTERS ..... 22

# 1. Introduction

The Security First Corp SecureParser is a robust library designed to secure digital information. This document describes the application programming interface, or API. It is aimed at the C developer for interfacing into applications.

## *1.1. Description*

The SecureParser is an M of N data splitting algorithm. Essentially, the algorithm uses cryptographic technique to redistribute a stream of data into a set of shares, such that no information may be obtained about the original data unless some set of M shares are recovered. When used in this mode, recovery is using any subset of M shares. SecureParser also allows for the creation of additional "mandatory" shares, which must always be available in order to permit data recovery, In addition, standard symmetric encryption and integrity checking can be externally invoked from the SecureParser as well.

## *1.2. Installation*

### 1.2.1. Disk Content

The SecureParser 4.0 release contains 3 files deployed in a ZIP file. These 3 files (capitalized for readability) are:

- **SETUP.EXE** – The executable setup program that invokes the windows installer program.
- **SETUP.INI** – Configuration parameters for the SETUP program
- **PARSER 4.0.MSI** – The windows installer program which installs the SecureParser toolkit.

### 1.2.2. Install

Extract the files to a directory on your system and run the SETUP.exe program to install. Once the installation is completed, the files listed below (capitalized for readability) will now reside in the destination directory.

- **LIBPARSER.DLL** – The SecureParser version 4.0, deployed as a DLL.
- **LIBPARSER.H** – The SecureParser version 4.0 API header file which allows for integration of external applications. This file contains functional type definitions.
- **LIBPARSER_TYPES.H** – The SecureParser version 4.0 types definition file which contains structure and variable definition.
- **LIBPARSER_KEYS** – Required for the UNITTEST program.
- **LIBEAY32.DLL** – The openSSL library version 0.9.7e which contains the encryption algorithm and the random number generator.
- **MSVCR71.DLL** – The Microsoft C Runtime Library version 7.10.3052.4
- **STDAFX.CPP** – The standard C++ library which allows for integration of C++ code to the SecureParser.

- **STDAFX.H** – Part of the standard C++ library.
- **UNITTEST.C** – The SecureParser 4.0 test harness source code.
- **UNITTEST.EXE** – The SecureParser 4.0 test harness executable, which executes 1000 passes through the SecureParser 4.0.
- **UNITTEST.VCPROJ** – The SecureParser 4.0 test harness Visual C++ project file.
- **README** – A text file containing this overview

### Secure Parser Test Harness:

Also installed and loaded to the Windows start menu is the SecureParser 4.0 test harness program, which can be executed from the programs menu by following the SecureParser / 4.0 test menu selection.

# 2. Data Types and Structures

## *2.1. Base Types, structures, and enumerators*

The SecureParser uses several data types and structures.

| Type | Description |
|---|---|
| Parser | Structure for holding the state and associated I/O data. |
| Key | Structure for holding a generic key. |
| Share | Structure for holding a generic buffer. |
| ParserParams | Structure for holding parser parameters. |
| HeaderInfo | Structure for holding header information. |
| FooterInfo | Structure for holding footer information. |
| EncContext | Structure for storing various encryption contexts. |
| SigContext | Structure for storing various signature contexts. |
| HashingContext | Structure for storing various hash contexts. |
| SplitContext | Structure for storing the split context. |
| ERROR_TYPE | Enumerator indicating a specific error type. |
| SPLIT_TYPE | Enumerator identifying the type of split algorithm to utilize. |
| ENC_TYPE | Enumerator identifying the encryption mode to use. |
| AUTH_TYPE | Enumerator identifying the authentication mode to use. |
| HASH_TYPE | Enumerator identifying the hashing mode to use. |
| TRANSFORM_TYPE | Enumerator identifying the transform type to use. |
| KEY_TYPE | Enumerator indicating a specific key type. |
| uint8 | An unsigned 8 bit integer. |
| uint16 | An unsigned 16 bit integer. |
| uint32 | An unsigned 32 bit integer. |

## *2.2. Enumerators*

### 2.2.1. ENC_TYPE

1. **ENC_NONE** – no encryption

2. **ENC_AES128CTR** – AES 128 bit in CTR mode
3. **ENC_AES192CTR** – AES 192 bit in CTR mode
4. **ENC_AES256CTR** – AES 256 bit in CTR mode
5. **ENC_AES128CBC** – AES 128 bit in CBC mode
6. **ENC_AES192CBC** – AES 192 bit in CBC mode
7. **ENC_AES256CBC** – AES 256 bit in CBC mode
8. **ENC_3DESCBC** – Triple DES in CBC mode

### 2.2.2. AUTH_TYPE

1. **SIGN_RSAPSS** – RSA-PSS algorithm
2. **SIGN_DSA** – Digital Signature Algorithm
3. **SIGN_ECSA** – Elliptic Curve Digital Signature Algorithm
4. **SIGN_HMACSHA1** –HMAC-SHA1 symmetric Message Authentication Code

### 2.2.3. HASH_TYPE

1. **HASH_NONE** – no hashing algorithm.
2. **HASH_SHA256** – SHA-256 hashing algorithm.

### 2.2.4. SPLIT_TYPE

1. **SPLIT_NONE** - No split is used.
2. **SPLIT_BLOCK** – Target Data will be split at an block level determined by the splitting algorithm
3. **SPLIT_BYTE** – Target Data will be split at the byte level
4. **SPLIT_BIT** – Target Data will be split at the bit level
5. **SPLIT_BIT_SCATTER** – Target Data will be bit scattered.

### 2.2.5. KEY_TYPE

1. **KEY_SYM** (keys for all symmetric algorithms)
2. **KEY_RSAPUB**
3. **KEY_RSAPRI**
4. **KEY_DSAPUB**
5. **KEY_DSAPRI**
6. **KEY_ECDSAPUB**
7. **KEY_ECDSAPRI**

### 2.2.6. ERROR_TYPE

1. **ERROR_NONE** – No error
2. **ERROR_NO_MASTER_KEY** – No master key, can not continue.
3. **ERROR_INVALID_CALLBACK** – An invalid callback was registered
4. **ERROR_INVALID_METHOD** – An invalid parse split method occurred.
5. **ERROR_NO_TARGET** – No target was set to parse.
6. **ERROR_NO_TARGET_LENGTH** – Target buffer has no length
7. **ERROR_NOT_ENOUGH_SHARES** – Unable to restore, not enough shares are present.
8. **ERROR_INVALID_SHARE** – An invalid share was set
9. **ERROR_INVALID_KEY** – An invalid key was set
10. **ERROR_INTEGRITY_CHECK** – An integrity check failure occurred
11. **ERROR_NO_MEMORY** – Out of memory

## *2.3. Data Structures*

### 2.3.1. Parser:

| Name | Type | Description |
|---|---|---|
| originalShare | Share | stores the original unsplit data. |
| shares | Share [] | an array of Share data structures of size MAX_SHARE_COUNT. |
| shareCount | int | actual number of shares used. |
| splitEncContext | EncContext * | buffer holding pre-encryption context between calls to doSplit and doRestore. |
| finalFlag | int | indicates that the final call to doSplit or doRestore will be made next (default = 0). |

### 2.3.2. ParserParams:

| Name | Type | Description |
|---|---|---|
| L | int | total number of mandatory shares (default = 0). |
| N | int | total number of non-mandatory shares (default = 0). |
| M | int | total number of non-mandatory shares needed for recovery (default = 0). |
| splitType | SPLIT_TYPE | the split algorithm to use (default = NONE). |
| splitEncMode | ENC_TYPE | the encryption mode to use during the split (default = NONE) |

| | | |
|---|---|---|
| | | NONE). |
| preEncMode | ENC_ TYPE | the pre-encryption mode to use (default = NONE). |
| postEncMode | ENC_ TYPE | the post-encryption mode to use (default = NONE). |
| splitHashMode | HASH_ TYPE | the hash mode to use (default = HASH_SHA256). |
| preAuthMode | AUTH_ TYPE | the pre-signing mode to use (default = NONE). |
| postAuthMode | AUTH_ TYPE | the post-signing mode to use (default = NONE). |
| preEncKey | char * | handle for the pre-encryption key (default = NONE). |
| postEncKey | char * | handle for the post-encryption key (default = NONE). |
| preAuthKey | char * | handle for the pre-signature key (default = NONE). |
| postAuthKey | char * | handle for the post-signature key (default = NONE). |
| maxOrigBufSize | int | max size for the original buffer (default = MAX_ORIG_BUF_SIZE). |
| maxShareBufSize | int | max size for each share buffer (default = MAX_SHARE_BUF_SIZE). |

### 2.3.3. KeyId:

| Name | Type | Description |
|---|---|---|
| id | char * | unique identifier for this key. |
| idLen | int | length of buffer holding id. |

### 2.3.4. KeyShare:

| Name | Type | Description |
|---|---|---|
| data | uint8 * | key share data for this share. |
| dataLen | int | length of data (in bits) of data buffer. |

### 2.3.5. Key:

| Name | Type | Description |
|---|---|---|
| keyId | KeyId | unique identifier for this key. |
| type | KEY_TYPE | type of key that this. |
| data | uint8 * | pointer to the buffer holding the key data. |
| dataLen | int | length of the buffer holding the key data. |

### 2.3.6. Share:

| Name | Type | Description |
|---|---|---|
| data | uint8 * | data buffer for this share. |
| dataLen | int | length indicating the amount of data found in this share. |
| encContext | EncContext | data buffer storing the encryption context between calls to doSplit and doRestore. |
| authContext | SigContext | data buffer storing the authentication context between calls to doSplit and doRestore. |
| hashContext | HashContext | data buffer storing the hash context between calls to doSplit and doRestore. |

### 2.3.7. EncContext:

| Name | Type | Description |
|---|---|---|
| encMode | ENC_TYPE | the type of encryption being used. |
| ... | Union | various types of encryption contexts can be placed here. |

### 2.3.8. EncContext_AESCBC

| Name | Type | Description |
|---|---|---|
| encKeyId | KeyId | unique identifier of the encryption key to use for this context. |
| currentIV | uint8 * | the current IV being stored for this context. |
| currentIdLen | int | the length of the current IV in bytes. |

### 2.3.9. EncContext_AESCTR

| Name | Type | Description |
|---|---|---|
| encKeyId | KeyId | unique identifier of the encryption key to use for this context. |
| currentIV | uint8 * | the current IV being stored for this context. |
| currentIVLen | int | the length of the current IV in bytes. |

### 2.3.10. AuthContext

| Name | Type | Description |
|---|---|---|
| authMode | AUTH_TYPE | the type of authentication being used. |
| ... | Union | various types of authentication contexts can be placed here. |

### 2.3.11. AuthContext_RSAPSS

| Name | Type | Description |
|---|---|---|
| pubKeyId | KeyId | unique identifier of the public key to use for this context. |
| privKeyId | KeyId | unique identifier of the private key to use for this context. |
| hashContext | openSSL? | Pointer to the OpenSSL hash context data type. |

### 2.3.12. AuthContext_DSA

| Name | Type | Description |
|---|---|---|
| pubKeyId | KeyId | unique identifier of the public key to use for this context. |
| privKeyId | KeyId | unique identifier of the private key to use for this context. |
| hashContext | openSSL? | Pointer to the OpenSSL hash context data type. |

### 2.3.13. AuthContext_ECSA

| Name | Type | Description |
|---|---|---|
| pubKeyId | KeyId | unique identifier of the public key to use for this context. |
| privKeyId | KeyId | unique identifier of the private key to use for this context. |
| hashContext | openSSL? | Pointer to the OpenSSL hash context data type. |

### 2.3.14. AuthContext_HMACSHA1

| Name | Type | Description |
|---|---|---|
| macContext | openSSL? | Pointer to the OpenSSL mac context data type. |

### 2.3.15. HashContext

| Name | Type | Description |
|---|---|---|
| hashMode | HASH_TYPE | the type of hashing being used. |
| ... | Union | various types of hashing contexts can be placed here. |

### 2.3.16. HashContext_SHA

| Name | Type | Description |
|---|---|---|
| hashContext | openSSL? | Pointer to the OpenSSL hash context data type. |

## 2.4. Data Format:

| Share Data |
|---|
| \n----- BEGIN POST-ENC KEY ID -----\n |
| post-encryption key id |
| \n----- END POST-ENC KEY ID -----\n |
| \n----- BEGIN POST-SIG KEY ID -----\n |
| post-signature key id |
| \n----- END POST-SIG KEY ID -----\n |
| \n----- BEGIN POST-ENC HEADER -----\n |
| post-encrypted header |
| \n----- END POST-ENC HEADER -----\n |
| \n----- BEGIN POST-ENC DATA -----\n |
| post-encrypted data |
| \n----- END POST-ENC DATA -----\n |
| \n----- BEGIN POST-ENC FOOTER -----\n |
| post-encrypted footer |
| \n----- END POST-ENC FOOTER -----\n |
| \n----- BEGIN POST-SIG DATA -----\n |
| post-signature data |
| \n----- END POST-SIG DATA -----\n |

| Post-encrypted header |
|---|
| \n----- BEGIN HEADER -----\n |
| Key share (len):key share information\n |
| Share id (2):share ID\n |
| L (2):L\n |
| M (2):M\n |
| N (2):N\n |
| Split mode (1):split mode\n |

| Split-encryption key (len):split-encryption key id\n |
|---|
| Split-encryption IV (len): split-encryption IV data\n |
| Split-hash mode (1):split-hash mode\n |
| Original buffer (4):original buffer size\n |
| Share buffers (4):share buffer size\n |
| Pre-encryption key (len):pre-encryption key id\n |
| Pre-signature key (len):pre-signature key id\n |
| \n----- END HEADER -----\n |

| **Post-encrypted footer** |
|---|
| \n----- BEGIN ENCRYPTED FOOTER -----\n |
| encrypted footer |
| \n----- END ENCRYPTED FOOTER -----\n |

| **Encrypted footer** |
|---|
| \n----- BEGIN SPLIT-HASH DATA -----\n |
| split-hash data |
| \n----- END SPLIT-HASH DATA -----\n |
| \n----- BEGIN PRE-SIG DATA -----\n |
| pre-signature data |
| \n----- END PRE-SIG DATA -----\n |

# 3. Function Library Reference

The function library provides all the necessary functions to initialize, parse, restore, and close the Parser context.

## 3.1. parser_createParser

**Parameters** *none*
**Returns** Parser *p
**Description** Allocates the SecureParser context, and sets default values.

## 3.2. parser_destroyParser

**Parameters** Parser *p
**Returns** void
**Description** Destroys the SecureParser context, including parser parameter objects and share buffers.

## 3.3. parser_createKey

**Parameters** char *id
uint8 *data
size_t length
KEYTYPE type
**Returns** Key *k
**Description** Constructs the structure **k** for use with the parser from the unque id, key data buffer, length indicater and key type given.

## 3.4. parser_destroyKey

**Parameters** Key *k
**Returns** *none*
**Description** Destroys the key pointed to by k. Note: Do not destroy a key currently in use with the Parser.

## 3.5. parser_createShare

**Parameters** int length
Key *encKey
Key *authKey
HASH_MODE h
**Returns** Share *s
**Description** Constructs a Share data structure based on the given length, the encryption key type, authentication key type and hash mode.

### 3.6. parser_destroyShare

| | |
|---|---|
| Parameters | Share *s |
| Returns | none |
| Description | Destroys the share pointed to by s, including all encryption and authentication keys. |

### 3.7. parser_initParameters

| | |
|---|---|
| Parameters | ParserParams *params |
| Returns | int error |
| Description | Initializes a ParserParams data structure to the default parametr set. The return error should be 0 for success, and non-zero error message for failure. |

### 3.8. parser_setParameters

| | |
|---|---|
| Parameters | Parser *p, ParserParams *params |
| Returns | int error |
| Description | Deletes any old ParserParams that may have already been added to the Parser struct. Creates a copy of the new ParserParams pointed to by params for internal use and stores in in the Parser pointed to by p. If params is altered or destroyed by the user, Parser p will still have retained a copy for itself that is not affected. Any data structures that must be created by the parameters struct, such as Shares or Keys will be created. The Parser will destroy the params object itself upon being closed. |

### 3.9. parser_setFinalize

| | |
|---|---|
| Parameters | Parser *p |
| Returns | none |
| Description | Sets the internal state of the Parser to finalize the output stream during the next SecureParser split session. After setFinalize is called, only one more call to doSessionSplit may be made. |

### 3.10. parser_getOriginalShare

| | |
|---|---|
| Parameters | Parser *p |
| Returns | Share *s |
| Description | Returns a Share structure representing the original buffer used by the Parser. |

### 3.11. parser_getShare

| | |
|---|---|
| Parameters | Parser *p,<br>int ShareID |
| Returns | Share *s |
| Description | Returns a Share containing the Share structure for the Share matching the given ShareID. |

### 3.12. parser_doSplit

| | |
|---|---|
| Parameters | Parser *p |
| Returns | int error |
| Description | Performs the SecureParser split. This action treats data that is in the target buffer as input and places split data that is output into buffers for each Share. A return value of 0 indicates success, otherwise a failure has occurred and the error is returned. If setFinalize has been called, this must be the last |

### 3.13. parser_doRestore

| | |
|---|---|
| Parameters | Parser *p |
| Returns | int error |
| Description | Performs the SecureParser restore. This action treats all data in share buffers as input and places output data (the reconstructed original data) into the target buffer. A return value of 0 indicates success, otherwise a failure has occured and the error is returned. |

### 3.14. parser_generateHeaders

| | |
|---|---|
| Parameters | Parser *p |
| Returns | int error |
| Description | Generates the necessary header information based on the parser's current parameter set and state, and puts this header information appropriately within each Share buffer. |

### 3.15. parser_generateFooters

| | |
|---|---|
| Parameters | Parser *p |
| Returns | int error |
| Description | Generates the necessary footer information based on the parser's current parameter set and state, and puts this footer information appropriately within each Share buffer. |

### 3.16. parser_restoreHeaders

| | |
|---|---|
| Parameters | Parser *p |

| | |
|---|---|
| Parameters | Parser *p |
| Returns | int error |
| Description | This function assumes that the data within each Share buffer is header information and will be restored into the Parser, its parameters reset and its state changed appropriately. |

### *3.17. parser_restoreFooters*

| | |
|---|---|
| Parameters | Parser *p |
| Returns | int error |
| Description | This function assumes that the data within each Share buffer is footer information and will be restored into the Parser, its parameters reset and its state changed appropriately. |

### *3.18. parser_getRestoreStatus*

| | |
|---|---|
| Parameters | Parser *p |
| Returns | int status |
| Description | This funcion will return the status of the prior restore process. Either a success value will be returned if the restore process has completed with no errors, or the appropriate error status. |

# 4. Function Details

## 4.1. *parser_generateHeaders*

### Purpose

The purpose of this function is to generate share header information for each share based on any meta-data that must necessarily be communicated to the SecureParser when later performing a restore algorithm. The header information can only contain data that is available to the SecureParser prior to the start of the data splitting process. The parser_generateHeaders routine creates the header blocks for each share and places them within each Share buffer. A call to parser_generateHeaders must be made prior to any calls to parser_doSplit.

### Algorithm

1. Split key using Shamir splitting algorithm.
2. Create a temporary buffer.
3. For each share...
    3.1. Include key share information in temporary buffer.
    3.2. Include share ID in temporary buffer.
    3.3. Include L, M and N in temporary buffer.
    3.4. Include split mode in temporary buffer.
    3.5. Include split-encryption key ID in temporary buffer.
    3.6. Include split-encryption IV in temporary buffer.
    3.7. Include split-hash mode in temporary buffer.
    3.8. Include original buffer size in temporary buffer.
    3.9. Include share buffer size in temporary buffer.
    3.10. Include pre-encryption key ID in temporary buffer.
    3.11. Include pre-encryption IV in temporary buffer.
    3.12. Include pre-signature key ID in temporary buffer.
    3.13. Include pre-signature IV in temporary buffer.
    3.14. Replace key share information in temporary buffer with Share i's key share.
    3.15. Replace share ID in temporary buffer with Share i's ID.
    3.16. Begin split-hash on the data in temporary buffer and store context for later.
    3.17. Encrypt temporary buffer with post-encryption key and post-encryption IV and include resulting ciphertext in Share i's output buffer. Maintain Share i's post-encryption context for later use.
    3.18. Begin post-signature on Share i's output buffer and store context for later.

## 4.2. *parser_doSplit*

### Purpose

The purpose of this function is to split all data from the target buffer into share buffers to be later handled by the calling function. A series of steps are taken to provide confidentiality, integrity, authenticity and data partitioning for the data that is split, before and after the data is split.

### Algorithm

1. Test if pre-encrypt is on.

    If Yes,

    1.1. Encrypt target buffer under pre-encrypt key and pre-encrypt iv.

2. Test if pre-sign is on.

    If Yes,

    2.1. Sign target buffer under pre-sign key.

    2.2. Test if finalize has been set.

        If Yes,

        2.2.1. Finalize pre-signature context.

3. Do split-encryption using split-encryption key and split-encryption context. (Should be inialized during the first call to this function.)

4. Call appropriate split algorithm...

5. Test if hashing is on

    If Yes,

    5.1. For each output share

        5.1.1. Continue hash for share i's buffer.

6. Test if post-encrypt is on.

    If Yes,

    6.1. For each output share.

        6.1.1. Encrypt share i's buffer using share i's post-encryption key and post-encryption context.

7. Test if post-sign is on

    If Yes,

    7.1. For each output share

        7.1.1. Continue signature for share i's output buffer under share i's post-signature key using share i's post-signature context.

## 4.3. *parser_generateFooters*

### Purpose

The purpose of this function is to generate share footer information for each share based on any meta-data that must necessarily be communicated to the SecureParser when later performing a restore algorithm. The footer information is typically only available to the SecureParser after the body of the data has be parsed. The parser_generateFooters routine creates the footer blocks for each share and places them within each Share buffer. parser_generateFooters must be called following all calls to parser_doSplit and parser_setFinalize.

### Algorithm

1. Create temporary buffer 1 for unencrypted footer information.
2. Create temporary buffer 2 for partially encrypted footer information.
3. For each share...
    3.1. Include pre-signature data in unencrypted buffer.
    3.2. Encrypt temporary buffer 1 using split-key and split-encryption context and include resulting ciphertext in temporary buffer 2.
    3.3. Continue split-hash over temporary buffer 2 using share i's split-hash context.
    3.4. Include split-hash data in temporary buffer 2.
    3.5. Encrypt temporary buffer 2 using share i's post-encryption key and post-encryption context and place resulting ciphertext in share i's output buffer.
    3.6. Continue post-authentication signature over share i's output buffer using share i's post-signature context.
    3.7. Finalize share i's post-signature context.
    3.8. Include share i's post-signature in share i's output buffer.

## 4.4. *parser_restoreHeaders*

### Purpose

The purpose of this function is to restore header information for each share based on any meta-data that was stored by the SecureParser for later restoration. The header information may contain data that is required for future calls to the parser_doRestore process to function properly. The parser_restoreHeaders routine requires that the header blocks for each share be placed within each Share buffer. A call to parser_restoreHeaders must be made prior to any calls to parser_doRestore.

### Algorithm

1. For each share...

    1.1. Create a temporary buffer for share i.

    1.2. Begin signature over share i's buffer using share i's post-signature key and post-signature IV. Maintain post-signature context for share i.

    1.3. Decrypt share i's buffer using share i's post-encryption key and post-encryption IV and place contents into share i's temporary buffer.

    1.4. Recover key share information for share i from share i's temporary buffer.

    1.5. Test if this share 0.

        If no,

        1.5.1. Compare share i's header information in temporary buffer with share 0's header information in temporary buffer. If there is a mismatch, return an error.

2. Recover split-encryption key, based on all key share information for in each share's temporary buffer.

3. Recover parser state based on header information found in share 0's temporary buffer.

## 4.5. *parser_doRestore*

### Purpose

The purpose of this function is to restore all shares to their original single source. A series of steps verify that the data has not been tampered with before and after data restoration has completed.

### Algorithm

1. Test if post-sign is on.

    If Yes,

    1.1. For each output share

        1.1.1. Sign share i's output buffer under share i's post-signature key.

2. Test if post-encrypt is on.

    If Yes,

    2.1. For each output share.

        2.1.1. Decrypt share i's buffer using key post-decrypt key i and IV i.

3. Test if hashing is on

    If Yes,

    3.1. For each output share

        3.1.1. Continue hash for share i's buffer.

4. Call appropriate restore algorithm...

5. Do split-decryption using split-decryption key and split-decryption context. (Should be inialized during the first call to this function.)

6. Test if pre-sign is on.

    If Yes,

    6.1. Sign original buffer under pre-sign key using pre-signature context or pre-signature IV.

    6.2. Test if finalize has been set.

        If Yes,

        6.2.1. Finalize pre-signature context.

7. Test if pre-encrypt is on.

    If Yes,

    7.1. Decrypt original buffer under pre-decryption key.

## 4.6. *parser_restoreFooters*

### Purpose

The purpose of this function is to restore footer information for each share based on any meta-data that was stored by the SecureParser for later restoration. The footer information may contain data that is needed for valid completion of the restore process. The parser_restoreFooters routine requires that the footer blocks for each share be placed within each Share buffer. A call to parser_restoreFooters must be after all calls to parser_doRestore and parser_setFinalize.

### Algorithm

1. Create a temporary pre-signature buffer.

2. For each share...

    2.1. Create a temporary buffer for share i.

    2.2. Continue post-signature over share i's buffer using share i's post-signature key and post-signature context.

    2.3. Finalize share i's post-signature context.

    2.4. Compare share i's post-signature result with expected post-signature result and return an error on mismatch.

    2.5. Decrypt share i's buffer using share i's post-decryption key and post-decryption context and place contents into share i's temporary buffer.

    2.6. Continue split-hash over share i's encrypted footer portion of share i's temporary buffer using share i's split-hash context.

    2.7. Finalize share i's split hash context.

    2.8. Compare share i's split-hash result with expected split-hash result found in remaining portion of Share i's temporary buffer and return an error on mismatch.

    2.9. Decrypt the footer portion of share i's temporary buffer using the split-decryption key and split-decryption context and place resulting plaintext into the temporary pre-signature buffer.

    2.10. Compare temporary pre-signature buffer with finalized pre-signature result and return an error on mismatch.

1 INTRODUCTION ..... 3

2 SECUREPARSER ARCHITECTURE—OVERVIEW ..... 4

3 PRELIMINARIES ..... 6

3.1 The Advanced Encryption Standard (AES) ..... 6

3.2 Cryptographic Hash Functions ..... 8

3.3 Message Authentication Codes ..... 8

3.4 Random Number Generation ..... 8

3.5 Digital Signatures ..... 9

3.6 Secret Sharing (Informative) ..... 9

3.7 The Package Transform (Informative) ..... 10

3.8 Hash Trees (Informative) ..... 10

4 PRE- AND POST-ENCRYPTION ..... 11

4.1 Pre-encryption. ..... 11

4.2 Post-encryption. ..... 11

4.3 Data Authenticity and Integrity. ..... 11

4.4 Message Authentication Codes. ..... 11

4.5 Meta-Data. ..... 12

4.6 AES Encryption ..... 12

5 CRYPTOGRAPHIC DATA SPLITTING ..... 14

5.1 Input Parameters ..... 14

5.2 Procedure for Data Split ..... 15

5.3 Procedure for Data Restore ..... 17

5.4 Signing and Share Integrity ..... 20

6 SECUREPARSER DATA PARTITIONING ALGORITHMS ..... 21

6.1 "BlockSplit" ..... 21

6.2 "BitSegment" ..... 22

6.3 "BitScatter" ..... 24

6.4 "BitSplit" ..... 26

7 CONCLUSION ..... 29

# 1 Introduction

This document contains a high-level description of the SecureParser v4.0 cryptographic core. The SecureParser is a high-security cryptographic module designed to protect the confidentiality and integrity of stored data. To accomplish this goal, the SecureParser provides data encryption, decryption, and authentication/integrity protection. Additionally, the SecureParser implements the technique of "data splitting" (also referred to as "parsing"), which efficiently divides input data into shares in a cryptographically secure manner.

The features described in this document include:

1. An overview of the SecureParser processing architecture, which provides for data processing operations such as pre-encryption, data splitting (parsing), post-encryption, authentication and data compression.
2. An authentication mechanism designed to protect data integrity and detect data tampering.
3. A data-splitting (parsing) algorithm which processes input data into multiple output "shares". Data splitting techniques have the property that no data is revealed unless some (pre-specified) number of shares are recovered.

The security mechanisms described in this document may be combined with other techniques, and do not preclude the implementation of other elements within the SecureParser framework, e.g., media-specific functionality or alternative security mechanisms.

The algorithms described in this document have three principal advantages over alternative approaches to splitting data:

- **High throughput.** Data processing involves AES encryption operations and XOR operations, and a limited amount of polynomial computation. Furthermore, processing time can be further improved by dividing work amongst several machines, processors, or hardware encryption circuits operating in parallel.
- **Industry-standard security**. The security of the constructions is based on the strength of standard cryptographic algorithms such as AES, HMAC-SHA1 and Shamir secret sharing.
- **Simplicity of implementation.** The algorithms presented require only a single encryption function (AES encryption), techniques for Shamir sharing, a MAC function and a cryptographically-secure Pseudo-Random Number Generator (CSPRNG).

# 2 SecureParser Architecture—Overview

The SecureParser system performs a variety of operations on input data in order to provide data confidentiality and integrity. In addition, the SecureParser architecture leaves room for other operations, e.g., compression and error-correction coding, or alternative methods of authenticating files. These operations are considered external to the SecureParser's trusted core, and may be implemented outside of the SecureParser itself.

The following architecture divides the SecureParser into three separate levels which contain separable functionality. Figure 1 below illustrates these levels, and provides examples of processing functions which might be performed within each.

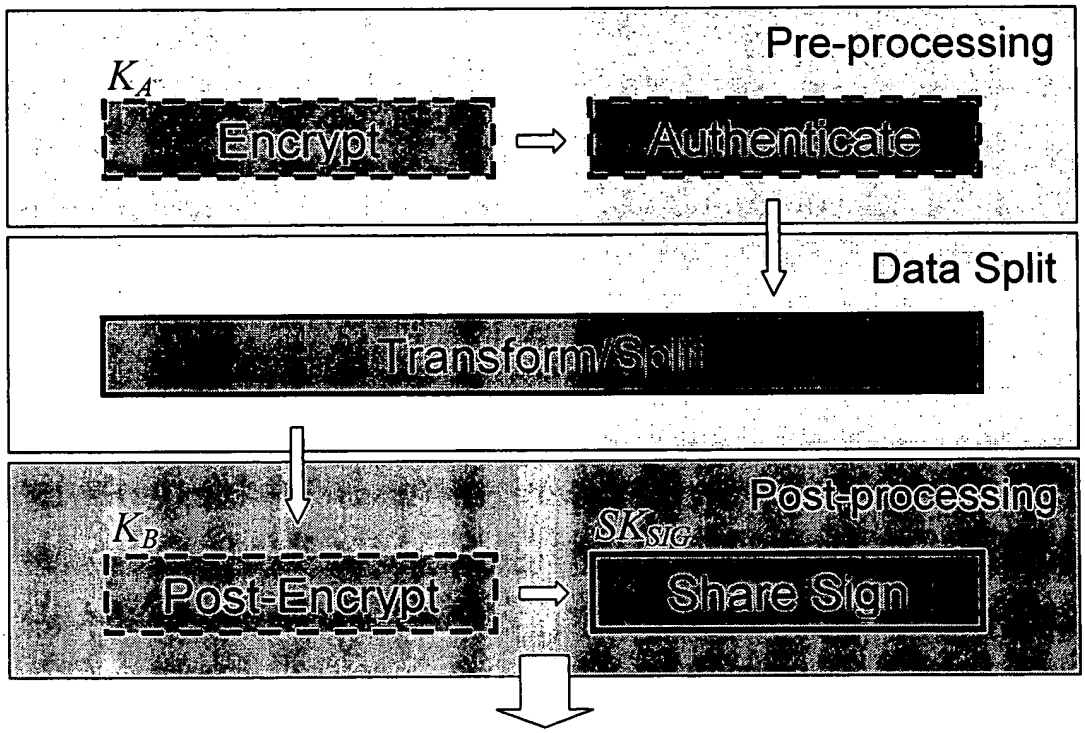



**Figure 1: Proposed architecture process-flow, from top-left. (Broken lines denote optional processes.)**

The proposed architecture divides data processing into three general stages:

1. **Pre-processing.** In this stage, data may be optionally pre-encrypted and authenticated using a secure authenticated encryption mode. This encryption is separate from the data-splitting algorithm, and serves as an optional layer of protection.
2. **Data split.** This stage contains the core of the SecureParser algorithm, and implements one or more of the data splitting algorithms proposed in a later section of this document.

3. **Post-processing.** In this stage, individual shares are authenticated (signed) using a standard signature algorithm. Optionally, file shares may be themselves encrypted using some external mechanism. After post processing, error correcting or detecting codes may be employed.

## 2.1 Data Splitting

Prior to discussing the specific mechanisms used to split data in the SecureParser, we briefly describe the security requirements of a data-splitting system. These goals should stand in the face of an adversary with detailed knowledge of the underlying algorithms, and access to some shares.

1. **Correctness.** Given any "legitimate" set of shares (e.g., *M* of *N*) created by the parsing algorithm, a user should be able to recover the initial file.
2. **Confidentiality.** An attacker should be unable to recover any information about an underlying plaintext (other than file size), without recovering the full set of shares required by the split.
3. **Randomness.** Two operations of the splitting algorithm should produce unlinkable, dissimilar outputs, even when executed on the same input.
4. **Integrity.** An attacker should be unable to alter any of the file shares without this alteration being detected.
5. **Modular Data Splitting.** Many different data splitting models should be admissible.

Optional application-specific goal:

6. **Bulk Protection of Data.** The amount of data that an adversary needs to access to reconstruct a file should be at least linear in the size of the file. This requirement motivates the second data splitting algorithm, which is based on the "all or nothing transform".

# 3 Preliminaries

This section briefly introduces the notation and cryptographic primitives used in the remainder of this document. Throughout this specification, variables are denoted as boldfaced letters, e.g., **X**, **Y**, **Z**. |**A**| indicates the size (in bits) of a variable. Concatenation of values is represented as (**A** | **B**), and the Exclusive-OR (XOR) operation as (**A** ⊕ **B**).

## 3.1 The Advanced Encryption Standard (AES)

The Advanced Encryption Standard (FIPS 197) is a NIST-approved block cipher designed to operate on fixed 128-bit data blocks, using a key size of 128, 192 or 256 bits. AES consists of two algorithms, **AES-ENC** and **AES-DEC**, which each accepts a key **K** and 128-bit input block, producing a 128-bit output. Use of these algorithms to directly encipher messages is typically referred to as "Electronic Codebook" mode (AES-ECB).

AES is approved by NIST to be used in several modes, including CBC, CTR and CFB. These modes provide for the secure encryption of longer messages, and allow for *randomized* encryption, where multiple encryptions of a given message generate distinct ciphertexts. The constructions presented in this document can be instantiated using any of the above modes. For the purposes of this implementation, we select CTR-mode encryption as described below.

### 3.1.1 AES-CTR

AES-CTR mode uses the **AES-ENC** primitive to implement a stream cipher. For every 128-bit plaintext block to be encrypted, CTR mode first generates a 128-bit keystream block by calling **AES-ENC (K, T)**, where **T** is a *counter value* incremented at each iteration (when **T** exceeds the maximum counter value, it wraps back to 0). The plaintext block is then encrypted by XORing it with the keystream block.

CTR-mode encryption is defined by the following piece of pseudo-code. **K** is an encryption key, and $\mathbf{T_0}$ is an initial counter value (Initialization Vector). The plaintext is represented as a series of 128-bit blocks ($\mathbf{M_1}$ , ... , $\mathbf{M_n}$) where $\mathbf{M_i}$ represents the $i^{th}$ block of plaintext. AES-CTR does not require plaintext padding: when the plaintext does not divide perfectly into an integer number of blocks, then the final block $\mathbf{M_n}$ may be a partial block of size **p** bits, where $\mathbf{p} < 128$. When a **p**-bit plaintext block is XORed with a normal keystream block, only the first **p** bits of the two blocks are considered.

The resulting ciphertext is represented as a series of blocks ($\mathbf{C_1}$ , ... , $\mathbf{C_n}$) along with the initial counter value $\mathbf{T_0}$. To encrypt a message using AES-CTR:

$\mathbf{T} = \mathbf{T_0}$
For i = 1 to n
    $\mathbf{C_i} = \mathbf{M_i} \oplus$ **AES-ENC(K, T)**
    $\mathbf{T} = \mathbf{T} + 1$

CTR-mode decryption is identical to encryption, except that the algorithm operates on ciphertext blocks. To correctly decrypt a CTR-mode ciphertext ($\mathbf{T_0}$, $\mathbf{C_1}$ , ... , $\mathbf{C_n}$), compute:

$\mathbf{T = T_0}$
For i = 1 to n
    $\mathbf{M_i = C_i \oplus AES\text{-}ENC(K, T)}$
    $\mathbf{T = T + 1}$

CTR mode encryption requires that the 128-bit initial counter value $\mathbf{T_0}$ be initialized to any value between 0 and $2^{128} - 1$ inclusive. The default value for $\mathbf{T_0}$ is 0. This counter is subsequently incremented for each 128-bit block encrypted or decrypted (if the counter exceeds $2^{128} - 1$, it wraps back to 0). To ensure the security of CTR-mode encryptions, it is critical that no value of **T** ever be re-used without re-keying. The method of choosing the initial counter is dependent on the implementation.

CTR mode may be parallelized across multiple processes or processors: after selecting an initial counter $\mathbf{T_0}$, the plaintext may be divided into separate segments and encrypted separately, provided that each encryption process is given the correct counter offset for the first block it encrypts.

### 3.1.2 AES-CBC

AES-CBC (Cipher-Block Chaining) mode uses the **AES-ENC** and **AES-DEC** primitives to encrypt/decrypt multi-block plaintexts. In addition to the encryption key **K**, CBC mode requires a 128-bit Initialization Vector (IV), which we denote by $\mathbf{T_0}$. Note that for security, this Initialization Vector must be 1) unique for each message encrypted, and 2) be generated using a cryptographically-secure Random Number Generator (see section below).

AES-CBC mode requires that the plaintext consist of an even number of AES blocks, i.e., is a multiple of 128-bits. When this is not the case, the plaintext should be padded by appending a 1 followed by as many 0s as necessary to reach a block boundary. If there is insufficient room to insert the shortest unambiguous padding sequence "10", an additional plaintext block must be added.

AES-CBC computes each ciphertext block by XORing a plaintext block with the previous ciphertext block, then encrypting the result using AES-ENC. In the case of the first ciphertext block, the Initialization Vector $\mathbf{T_0}$ replaces the previous ciphertext value. CBC-mode encryption is defined by the following piece of pseudo-code. The plaintext is represented as a series of 128-bit blocks ($\mathbf{M_1}$ , ... , $\mathbf{M_n}$) where $\mathbf{M_i}$ represents the $i^{th}$ block of plaintext. The resulting ciphertext is represented as a series of blocks ($\mathbf{C_1}$ , ... , $\mathbf{C_n}$) along with the Initialization Vector $\mathbf{T_0}$. To encrypt a message using AES-CBC:

$\mathbf{C_1 = AES\text{-}ENC(K, M_1 \oplus T_0)}$
For i = 2 to n
    $\mathbf{C_i = AES\text{-}ENC(K, M_i \oplus C_{i-1})}$

CBC-mode decryption is defined below. To correctly decrypt a CBC-mode ciphertext $(\mathbf{T_0}, \mathbf{C_1}, \ldots, \mathbf{C_n})$, compute:

$\mathbf{M_1} = \mathbf{AES\text{-}DEC(K, C_1)} \oplus \mathbf{T_0}$
For i = 2 to n
$\mathbf{M_i} = \mathbf{AES\text{-}DEC(K, C_i)} \oplus \mathbf{C_{i-1}}$

## 3.2 Cryptographic Hash Functions

The SecureParser integrity checking mechanisms require a collision-resistant cryptographic hash function. Hash functions accept an arbitrary-length input **M** and output a fixed-length hash value, and are strongly collision resistant. Except where otherwise specified, all hash functions in the SecureParser are implemented using SHA-256 (FIPS PUB 180-2), which accepts inputs of up to $2^{64}$ bits and outputs a 256-bit hash value.

## 3.3 Message Authentication Codes

Message Authentication Codes allow for the detection of unauthorized modifications to a message, which might occur if an adversary tampers with a stored file. MAC algorithms accept an arbitrary-length message **M** and a key $\mathbf{K_{MAC}}$, and output a fixed-length code which is typically stored with the message. Verifiers who have access to $\mathbf{K_{MAC}}$ can check this code by re-computing the MAC on the message, and comparing the result to the stored code. Throughout this document, Message Authentication Code computation is represented by the generic notation $\mathbf{MAC(K_{MAC}, M)}$.

All MAC functions in the SecureParser are implemented by HMAC-SHA1 (FIPS PUB 198) which produces a 160-bit MAC output. FIPS guidelines require a minimum key size of 80 bits, though for the purpose of this implementation we require a minimum of 128 bits, and recommend a key size of 160 bits.

A secure MAC can be combined with an appropriate encryption algorithm in order to provide "authenticated encryption". When these algorithms are combined, encryption will be performed first, and the MAC is computed on the resulting ciphertext (including any other information which will be stored/transmitted with the ciphertext, and used during the decryption process, e.g., the Initialization Vector.)

## 3.4 Random Number Generation

Many of the algorithms described in this document require a source of unpredictable bits that are statistically indistinguishable from truly random bits. Examples include key generation and the creation of Initialization Vectors. These values are generated using a deterministic cryptographically-secure pseudo-random number generation algorithm (CSPRNG), which must itself be correctly seeded from a suitable source of entropy such as a non-deterministic Random Number Generator. SecureParser implementations contain an internal CSPRNG implementation satisfying the requirements of FIPS 140-2, though correct seeding still falls to the host implementation.

Additionally, the SecureParser can be configured to work with an external RNG/PRNG provided by the host implementation. However, it is critical to note that flaws in the external (P)RNG or seeding procedure may compromise the functioning security of the SecureParser algorithms.

## 3.5 Digital Signatures

For purposes of share integrity and authentication, shares may be signed with an appropriate digital signature algorithm. The Secure Parser implements signatures using EC-DSA and DSA according to the guidelines of FIPS PUB 186-2. Digital signatures provide integrity protection and non-repudiability for stored information. Signatures allow re-constructing parties to test for modifications or damage to shares prior to conducting the re-construction process.

Signatures use public key cryptographic techniques, and possess two key components: a secret *signing key*, which is used only when authoring shares (e.g., during the splitting process), and a public *verification key*, which is used to validate the authenticity of signed shares. The signing key is kept secret by the splitting party, while the verification key may be published or stored in an appropriate manner.

When signatures are performed on split files, the signature is taken over the encrypted share, along with all meta-data: this includes a random per-file identifier that binds shares together; a share identifier, and a constant that defines the type of splitting that was performed along with other information used in the splitting process.

## 3.6 Secret Sharing (Informative)

The technique of "secret sharing", sometimes referred to as "split knowledge", allows parties to divide some piece of secret data into *shares* such that the secret can only be recovered by re-combining all, or some preset number, of the shares. A simple $n$-part split of a secret S can be accomplished in a secure manner using a cryptographically-secure pseudo-random number generator and the XOR operation as follows:

$$S_1 = \text{RAND}(|S|), \ldots, S_{n-1} = \text{RAND}(|S|),$$
$$S_n = S \oplus S_1 \oplus \ldots \oplus S_{n-1}$$

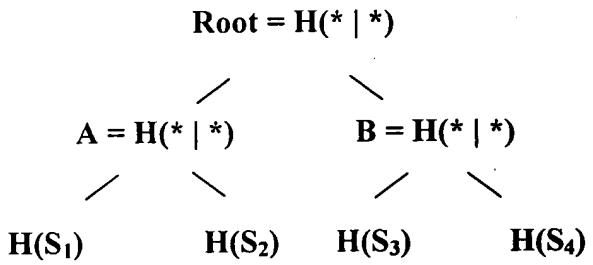
The technique developed by Shamir in 1979 allows for *threshold* splitting, in which the number of shares required to recover the secret ($t$) may be smaller than the total number of shares ($n$). This is generally referred to as $t$-out-of-$n$ secret splitting, where $t \leq n$. The Shamir algorithm encodes a secret as the y-intercept of a polynomial of degree $t$-1, and computes each secret share as the coordinates of some other point on the curve, computed over a finite field. Given any $t$ shares, it is possible to interpolate to find the y-intercept and thus recover the secret. The Shamir scheme has an important property for secure implementations: it provides *information-theoretic* ("perfect") security. Even when an attacker obtains $t$-1 shares, she gains no information about the secret, as every possible value represents a valid solution to the equation formed by the $t$-1 shares.

## 3.7 The Package Transform (Informative)

The "package transform" (Rivest, 1997) is a randomized algorithm which adds "security through bulk" for data splitting. In the Rivest technique, data is first encrypted using a secure block cipher under a randomly-generated session key **K**. Each |**K**|-bit block of the resulting ciphertext is then XORed with the session key. This process may be reversed by any party who obtains the obfuscated key and the entire ciphertext. However, it prevents recovery of the session key (and hence the ciphertext) by any person who does not possess all of the ciphertext blocks. Used correctly, this can prevent ciphertext recovery by an attacker who obtains a partial share.

## 3.8 Hash Trees (Informative)

Hash trees provide an efficient means to "commit" to large set of values or messages, using minimal storage. Hash trees are constructed using a collision-resistant cryptographic hash function—represented by **H()**—by computing the hash of a set of messages, e.g., file shares $\{S_1 \ldots S_N\}$, then recursively hashing pairs of hash results until only one "root" value remains. The following diagram illustrates a hash tree containing 4 elements.




Any party can verify the integrity of a share $S_1$ given only one value from each level of the tree, in this example $H(S_2)$, **B**, **Root** by computing $H( H((H(S_1) \mid H(S_2)) \mid B )$ and comparing the result to **Root**.

This technique is used to ensure the integrity of shares when only a subset of shares is available.

# 4 Pre- and Post-encryption

The SecureParser provides strong encryption for stored data, using the Advanced Encryption Standard algorithm. The Pre- and Post encryption layers are designed to operate independently from the data-splitting core, and provide an additional layer of confidentiality and integrity-checking for stored files. Data encryption in the SecureParser is designed to comply with FIPS 140-2 standards for symmetric-key encryption.

## 4.1 Pre-encryption.

Input files may be encrypted prior to execution of the data split procedure, in which case the encryption is termed *pre*-encryption. Pre- encryption operates on the input file as a whole, and produces a ciphertext which is then sent to the data-splitting core (if data splitting is being performed) to be broken into shares. This initial ciphertext includes authenticity/integrity-checking information, which is generated using a secret key held by the encryptor.

## 4.2 Post-encryption.

Subsequent to the splitting process, the split shares may themselves be encrypted, under one key (or several different keys). This process is termed *post*-encryption. Post-encryption operates on each data share as a completely independent file to be encrypted. Each of these the resulting post-split ciphertexts is embedded with its own integrity/authenticity information.

## 4.3 Data Authenticity and Integrity.

To ensure the authenticity of stored data, and detect malicious tampering or data storage errors, all encrypted data is authenticated using a Message Authentication Code (MAC) and optionally a digital signature to allow public verification. This protection ensures that any modifications to the data contents will be detected during the decryption process.

## 4.4 Message Authentication Codes.

Are generated by a symmetric-key algorithm, which requires a single secret key be present during both authentication *and* verification. This key should be stored along with the message decryption key, in an appropriate manner. Signatures are generated using a secret *signing key*, which must also be stored using appropriate security procedures. The corresponding signature *verification key* can be publicly distributed. However, its authenticity must be guaranteed using appropriate mechanisms, e.g., by incorporating it into a certificate which is then signed by a trusted authority.

## 4.5 Meta-Data.

Any meta-data describing the file is pre-pended to the file prior to encryption. This includes:

File name

File length

Creation/Modification dates

Any other host-specific file information

The resulting encrypted file should not be named for the original plaintext, in order to prevent information leakage from filenames, extensions, etc. To prevent this, the SecureParser includes a default-naming scheme for new encrypted files which uses a random number generator to create unique per-file encryption names.

## 4.6 AES Encryption

Data encryption in Secure Parser uses the AES algorithm (FIPS 197) in either of two NIST approved modes: AES-CBC and AES-CTR (see above for definitions). Each of these algorithms accepts key sizes of 128, 192 and 256 bits. Every file/share encrypted using a given key **K** requires a unique 128-bit Initialization Vector (or Initial Counter value). There are various constraints on the generation of these values (see the previous section), and correct generation is described below.

### 4.6.1 Key Generation

AES keys are used both to encrypt and decrypt data. Keys must be stored securely when not in use, either locally at the device, or on a removable tamper-resistant key-storage device. The SecureParser includes facilities for generating AES encryption keys, using a correctly seeded CSPRNG. While in host memory, keys encryption keys are protected in order to prevent them from being moved to other memory locations, or paged to disk, and they are securely deleted from memory when no longer in use. The mechanisms for ensuring memory protection are specific to the host implementation.

For each AES encryption key generated (of length 128, 192 or 256 bits), the key generation algorithm also generates a random MAC key of length 128-256 bits. This value is stored securely along with the encryption/decryption key, and the two are not separated.

### 4.6.2 Initialization Vector Generation

Secure encryption under AES-CBC or AES-CTR requires the generation of a 128-bit Initialization Vector or Initial Counter value (IV) for each message processed. This value is denoted as $\mathbf{T}_0$. The requirements for this value are different depending on which encryption mode is being used. In either case, the value is generated by computing 128 bits randomly from a CSPRNG for each message. This random value makes up the first 128-bits of the resulting ciphertext.

### 4.6.3 Encryption Padding

When using AES-CBC, plaintexts are padded using the padding scheme described in the definition of AES-CBC encryption above. This padding consists of a "1" value followed by one or more zeros, and must be sufficient to make the total message size equal to an integer multiple of the AES block size (128 bits). When there is insufficient room to include the shortest possible unambiguous padding sequence "10", an additional 128-bit block of "0"s is appended to the plaintext.

The AES-CTR algorithm does not require padding, and plaintexts may be of arbitrary size, not necessarily a multiple of the block size.

### 4.6.4 Encryption and Formatting

SecureParser encrypts data using the encryption mode specified by the user (AES-CBC/CTR), and pre-pends the Initialization Vector to the resulting ciphertext. The encryption also pre-pends an un-encrypted meta-data block which consists of the ciphertext length, encryption mode, and an optional random identifier for the file and/or for the encryption key. File and key identifiers are determined by the integration layer, and **are not** based on values from the file or key themselves.

When pre-encrypting data for a split procedure, the ciphertext produced by the pre-encryption stage is passed directly to the splitting core as it is generated. Similarly, the post-encryption process may begin to encrypt data that is being simultaneously generated by the splitting core.

### 4.6.5 Message Authentication/Signing

Once a file or data share has been completely encrypted using either AES-CBC or AES-CTR, the resulting ciphertext—including the meta-data block and the Initialization Vector—is authenticated using a Message Authentication Code generated under the appropriate secret MAC key. The SecureParser uses HMAC-SHA1 to compute MAC values on the ciphertext. The resulting 160-bit MAC tag is appended to the ciphertext.

When a user provides a secret signing key, files may also be signed using EC-DSA or DSA. The signing process should be applied across the ciphertext, the meta-data block, the Initialization Vector, and any MAC tag applied to the message

### 4.6.6 Verification and Decryption

Prior to decrypting any ciphertext, the file's authenticity is be checked by computing the MAC of the meta-data, ciphertext, and IV under the secret MAC key. This value should be compared to the MAC tag which was appended to the end of the ciphertext. Any mismatch in these values results in the immediate termination of the decryption process.

If this process succeeds, the resulting ciphertext is decrypted using the appropriate encryption key, and the plaintext along with the encrypted file meta-data is used to reconstitute the original file.

# 5 Cryptographic Data Splitting

The SecureParser incorporates the technique of secret sharing of Shamir (Adi Shamir, 1979) for encryption key security. This data is first encrypted using an authenticated encryption construction under a randomly-generated session key **K**. This key is subsequently "split" into *N* shares using the Shamir technique, such that any *M* shares are required in order to re-construct the original key. The ciphertext is then partitioned into *N* subfiles using one of several data splitting methods, and one key-share is pre-pended to each. The resulting construction offers rapid splitting and provable security.

Because security is achieved through the AES encryption and Shamir key-split, this approach permits a variety techniques for partitioning the resulting ciphertext, provided that the key shares are correctly generated and distributed. Several algorithms are defined below, and additional splitting algorithms may be defined in later documents.

## 5.1 Input Parameters

The three principal adjustable parameters provided to the splitting algorithm *L, M, N* are defined as follows:

- *L* refers to the number of mandatory shares.
- *M* is the number of non-mandatory shares required (along with the *L* mandatory shares) to recover the original file.
- *N* refers to the total number of non-mandatory shares. (Thus, there are a total of *N+L* shares produced.)

## 5.2 Procedure for Data Split

The full algorithm for splitting an input file consists of six stages:

1. **Key Generation.** In this stage, a session key **K** is generated using a secure PRNG.
2. **Data Encryption.** The plaintext is encrypted using AES-CTR or AES-CBC under the key **K**.
3. **Package Transform *(optional).*** The key **K** is processed using the Package transform, by XORing it with every |**K**|-bit block from the encrypted file. This approach prevents key recovery by an adversary who only recovers partial file shares.
4. **Key Secure.** The key **K** (which may have been obscured in the previous step) is divided into key-shares using the Shamir technique. Each share is distributed to one output file share. Alternatively, the key may be passed to an external secure key store.
5. **Parse/Distribute.** The ciphertext is partitioned into segments as defined by the split parameters. When fault-tolerant mode is used, this implies that the full ciphertext should be recoverable from **all** $L$ mandatory shares plus **any** $M$ non-mandatory shares). This data is stored into the file shares, along with the key shares generated in step 3.
6. **Share Integrity Protection.** Finally, the split process embeds integrity information into each share, which allows for the detection of damaged or malformed shares upon file recovery. This information is generated using cryptographic hashes.

Each of these steps is described in detail below.

### Step 1 - Key Generation

The first phase of the splitting operation requires the generation of a key **K** using a cryptographically secure RNG/PRNG, as defined elsewhere in this document. For AES, **K** may be 128, 192 or 256 bits.

### Step 2 - Data Encryption

Encryption is implemented using either AES-CBC or AES-CTR as specified by the user. Encryption under **K** requires a randomly generated Initialization Vector (Initial Counter) which must be generated using a RNG/PRNG.

**Plaintext padding.** First, pad the plaintext until it is equal in size to an integer number of AES blocks. This padding consists of a 1-bit followed by 0 bits. In some cases, this may involve the addition of one extra block of plaintext.

**Encryption.** To compute the encryption of a padded plaintext **M** :

1. Generate a random Initialization Vector $\mathbf{T_0}$ using a CSPRNG.
2. Encrypt the plaintext **M** under the key **K** using AES-CBC or AES-CTR using the IV $\mathbf{T_0}$ to produce **C**.

3. Prepend the value $\mathbf{T_0}$, to produce the ciphertext $\{\mathbf{T_0} \mid \mathbf{C}\}$.

### Step 3- Package Transform *(optional)*

To achieve security through bulk, the key **K** may optionally be processed using the Package Transform. **The Package Transform must be used in combination with the BlockSplit algorithm.** This security procedure prevents key recovery by an adversary who recovers only a partial share. To compute this transform:

1. Treat the ciphertext $\{\mathbf{T_0} \mid \mathbf{C}\}$ as $n$ **|K|-bit** blocks $(\mathbf{C_1}, \ldots, \mathbf{C_n})$.
2. Compute the new value of **K** as $\mathbf{K} = \mathbf{C_1} \oplus \ldots \oplus \mathbf{C_n} \oplus \mathbf{K}$.

Once the session key has been processed in this way, the original value of **K** cannot be recovered without access to all $n$ blocks of the file. Should an attacker lack all of any one of these blocks, she will be prevented recovering this key.

The use of this transform should be indicated in all resulting file shares, in order to allow correct recovery.

### Step 4- Key Split

This step uses the Shamir technique to split **K** across the mandatory and non-mandatory shares:

1. If the number of mandatory shares ($L$) is zero, set $\mathbf{J} = \mathbf{K}$ and start at step 3.
2. Otherwise, "split" into **K** into $L+1$ key-shares using the Shamir algorithm (or by using an XOR-based *n-of-n* split) such that *all* $L$+1 key-shares are required in order to reconstruct **K**. Distribute one of the first $L$ key-shares $(\mathbf{K_1}, \ldots, \mathbf{K_L})$ to each of the $L$ mandatory data-shares. Set $\mathbf{J} = \mathbf{K_{L+1}}$.
3. Next, split the value **J** into $N$ shares $(\mathbf{J_1}, \ldots, \mathbf{J_N})$ such that any $M$ shares are required in order to recover **J**. Distribute one of these $N$ shares to each of the non-mandatory data-shares.

Finally, the session key **K** should be securely wiped from memory.

### Step 5- Data Partition

Once data has been encrypted using the above operations, the resulting ciphertext $\{\mathbf{T_0} \mid \mathbf{C}\}$ must be partitioned and distributed into $L+N$ data shares. For such a partition to be correct, it must be possible to recover the full ciphertext from a share set consisting of **all** $L$ mandatory data-shares, along with **any** $M$ non-mandatory data-shares.

Before any data is partitioned, a "meta-data" block is pre-pended to the ciphertext to indicate the file length, encryption mode, key size, and a flag indicating whether the Package transform is to be used in processing the key **K**.

Data partitioning can be performed using a variety of algorithms. Unless the Package Transform option is being used, the partitioning algorithm need is not required to provide any security properties, as the encryption and key-splitting procedures provide confidentiality for the data. For brevity, three algorithms are presented in the Appendix

of this document: *BitSplit*, *BitScatter* and *BlockSplit*. **The Package Transform must be used in combination with the BlockSplit algorithm.**

### Step 6- Share Integrity Protection

The Integrity Protection process inserts additional information into file shares, which allows verifiers to detect errors prior to reconstructing a file.

The principal function of the Integrity Protection/Verification mechanism is to detect of non-malicious (accidental) damage to files, some of which might occur due to storage or communication errors. Integrity Protection is *not* designed to ensure the correctness of shares in the face of a deliberate attack, as an attacker may alter a share and also change its integrity information. To provide strong authenticity protection, it is recommended that users also sign shares under an appropriate secret signing key.

Integrity Protection can augment a separate authenticity check, as each share embeds cryptographic information about all of the other shares. When any one share in the recovery set is *known* to be authentic (e.g., is correctly signed), Integrity Protection allows recovery agents to detect flaws in the remaining shares by evaluating integrity-check information in the authentic share.

Share Integrity information is computed using a collision-resistant cryptographic hash function **H()** (implemented by SHA-256), using hash-tree construction. For a set of shares $\{\mathbf{SHARE_1}, \ldots, \mathbf{SHARE_{N+L}}\}$ (constructed according to the above process), integrity protection is computed as follows:

1. If $\mathbf{N+L} < 8$, append the value $(\mathbf{H(SHARE_1)} \mid \ldots \mid \mathbf{H(SHARE_{N+L})}$ to each of the shares.
2. If $\mathbf{N+L} \geq 8$, compute a hash tree from the share set:
    a. Compute each consecutive pair of share hashes pre-pended with the one-byte "0" value (e.g., $\mathbf{H(0x00 \mid SHARE_1)}$, $\mathbf{H(0x00 \mid SHARE_2)}$).
    b. Next, compute **H (0xFF | A | B)** where 0xFF is a one-byte value consisting of 8 "1" bits, **A** is the first hash result, **B** is the second hash result, and | is concatenation. If the total number is odd, compute **H (0xFF | A)** where **A** is the final share.
    c. Repeat step (a) on the resulting list, until only one hash result is left. This is the *root* of the hash tree.
    d. To each share $\mathbf{SHARE_i}$, distribute: $\mathbf{H(SHARE_i)}$, the tree root, along with up to one value from each level of the tree, as necessary to compute the tree root from $\mathbf{H(SHARE_i)}$.

## Procedure for Data Restore

The full algorithm for recovering an input file from shares consists of five stages:

1. **Share Integrity Verification.** All available shares are compared to verify data integrity, and to detect mismatches between share integrity information.
2. **Key Recovery.** The key **K** is recovered from the key-shares included with the *L* mandatory shares and any *M* non-mandatory shares.

3. **Data De-Partition.** The full ciphertext is reconstructed from the file shares.
4. **Inverse Package Transform.** If it was employed during the splitting process, the Package Transform process is inverted in order to reveal the original (un-processed) value of **K**.
5. **Data Decryption.** The ciphertext is decrypted under the key **K**.

Each of these steps is described in detail below.

### Step 1- Share Integrity Verification

Before recovering data from any set of shares, the recovery process verifies the integrity of all shares in the recovery set. By itself, this process does not ensure the authenticity of shares (see "Share Signing and Integrity" below), but will detect non-malicious damage to files. Additionally, if one share in the recovery set is *known* to be authentic, the share integrity procedures will detect even deliberate attacks on the remaining shares.

Share integrity verification proceeds using the hash values embedded during the data split process, as follows:

1. For each $\mathbf{SHARE_i}$, compute the cryptographic hash of the share (excepting the integrity section) pre-pended with a zero byte 0x00. Compare the result to the hash value $\mathbf{H\ (0x00\ |\ SHARE_i)}$ embedded in the share during the share construction process. In the event of a mismatch, abort recovery.
2. If the total number of shares $(N+L)$ is less than 8, check the hash of each share pre-pended with 0x00 against the corresponding value $\mathbf{H\ (0x00\ |\ SHARE_i)}$ stored with every other share. If any of these checks fails, reject the share(s) in question and abort recovery.
3. If the total number of shares $(N+L)$ is greater or equal to 8, then for each share $\mathbf{SHARE_i}$, compute the root of the hash tree using the information stored along with the share (see the share integrity section). If these values differ between shares, abort recovery.

### Step 2- Key Recovery

The second phase of the recovery operation reconstructs the value **K** from a legitimate set of mandatory and non-mandatory file shares. Each file share contains one key share, processed according to the key splitting algorithm described in the previous section.

To recover **K** from these file shares:

1. Combine any $M$ key-shares from the non-mandatory file-shares in order to recover the value **J**.
2. If no mandatory shares exist, set **K** = **J** and output **K**.
3. Otherwise, combine all of the key-shares from the mandatory file-shares ($\mathbf{K_1}$, ..., $\mathbf{K_L}$) with the final share **J**. Output the result as **K**.

### Step 3- Data De-partition

Data is subsequently recovered from the $L + M$ shares using the appropriate de-partitioning algorithm. All file shares are marked with an identifier indicating the

partitioning algorithm which was used during the data split process. De-partitioning algorithms are specified in the appendix.

The recovery process should reconstruct the ciphertext ($\mathbf{T_0} \mid \mathbf{C}$), and recover the meta-data block indicating file size, key length, encryption mode, and use of the package transform.

### Step 4- Inverse Package Transform

If the meta-data associated with the shares indicates that the package transform was used during the split process, the inverse process must be applied to the value **K**. The package transform process can be inverted using the same algorithm used to apply the original transform:

1. Treat the ciphertext **C** as $n$ **|K|-bit** blocks ($\mathbf{C_1}$ , ... , $\mathbf{C_n}$).
2. Compute the new value of **K** as $\mathbf{K} = \mathbf{C_1} \oplus ... \oplus \mathbf{C_n} \oplus \mathbf{K}$.

### Step 5- Data Decryption

Next, decrypt the ciphertext **C** under the recovered key **K** using AES-CTR or AES-CBC as indicated by the meta data included in the shares, using the Initialization Vector (Initial Counter) value $\mathbf{T_0}$. This recovers the plaintext **M**. Discard the encryption padding (recognizable as a 1 followed by a series of 0s).

Output **M** as the result of the data recovery process.

## 5.4 Signing and Share Integrity

It is important to protect against share tampering, which might occur if an attacker gains control of a storage channel. Without adequate protections, tampering may allow a sophisticated attacker to modify the shares so that the re-constructed file is incorrect.

The Share Integrity mechanism (described above) does not by itself guarantee the authenticity of file shares. However, provided that any one share is known to be authentic, the integrity-checking mechanism *can* detect unauthorized modifications any of the other shares.

The recommended mechanism for ensuring authenticity/integrity is to sign the Integrity segment of one or more file shares using a secure digital signature algorithm. The authenticity of this data can be verified against the corresponding verification key, and the remaining shares can be verified from the Integrity segment itself.

Share signing in the SecureParser is achieved using either the EC-DSA signing algorithm using 384/512-bit curves as defined in FIPS 186-2, or the standard DSA algorithm using a modulus of size 512-1024 bits. Signing keys must be generated using the appropriate algorithm, and should be stored securely in compliance with FIPS-140 standards. Verification keys should also be certified according to those standards, and should be signed by some trusted authority.

# 6 SecureParser Data Partitioning Algorithms

## 6.1 "BlockSplit"

When a file is processed using the optional Package Transform, the resulting ciphertext must be partitioned divided into shares such that following properties hold over the resultant shares. We say that a set of shares is *legitimate* if it is allowed to reconstruct the data, and that it is *illegitimate* if it is not allowed to reconstruct the data.

1. Any set of shares which contains some portion of every output block ($\mathbf{T_0}$, $\mathbf{C_1}$, ... , $\mathbf{C_n}$) is legitimate.
2. Every legitimate set of shares contains all of every output block.

Provided that the above requirements are met, this split may be accomplished using any technique that supports the desired policy. The "BlockSplit" algorithm is one such technique, and is defined below:

**Notation**. In the description of this algorithm, we standardize a number of definitions.

- A *block* refers to a consecutive group of bits whose size equals the |**K**| (i.e., either 128, 192, or 256). (This should not be confused with the AES block size, which is always 128 bits.)
- A *segment* is an abstraction consisting of groups of blocks; an entire file will be divided into segments for the splitting and joining processes.
- [r] indicates the greatest integer not exceeding r
- C (n,k) denote the binomial coefficient "n choose k," the number of k-element subsets of a set of n elements, equal to $n!/(k!(n-k)!)$.
- Note also that all counting, whether of blocks, segments, or shares, is zero-indexed.

**Assigning Segments.** A splittable file must contain at least $L+C(N, M-1)$ segments; any smaller number of segments will allow the file to be recovered with fewer than $L+M$ shares. Each segment must contain at least one complete block.

The blocks are then divided up into segments as follows. Let F be the number of blocks in the given file, and define:

$$a = \left\lfloor \frac{F}{\binom{N-1}{M-1} \cdot \left( \frac{N}{N-M+1} + L \right)} \right\rfloor = \left\lfloor \frac{F}{\binom{N}{M-1} + \binom{N-1}{M-1} \cdot L} \right\rfloor, b = a \cdot \binom{N}{M-1}.$$

Then the $i^{th}$ segment (counting from zero) contains blocks (a * i) through (a $(i+1)$-1), inclusive, if $i < C(N,M-1)$; otherwise, the remaining L segments are partitioned evenly from the end of the file and contain blocks b + [$(F - b)$ * (i – $C(N,M-1)$) / $L$] through b +

$[(F - b) * ((i+1) - C(N,M-1)) / L] - 1$, inclusive. These formulae result in the final output shares being equally distributed in size.

**Assigning segments to shares**. The last $L$ segments of the file are assigned in order to the $L$ mandatory shares, with one segment per share. To assign the first $C(N, M-1)$ segments to the $N$ non-mandatory shares, we first enumerate all $C(N, M-1)$ of the $(M-1)$-element subsets of {0, 1, ..., N-1} in lexicographical order. If (and only if) the $i^{th}$ such subset does not contain the element $j$, then the $j^{th}$ non-mandatory share will contain segment $i$. This ensures that every possible collection of $M-1$ non-mandatory shares is missing at least one of the $C(N, M-1)$ segments, while every collection of $M$ non-mandatory shares has all of them, and this assignment is optimal.

**Writing out the shares**. The $i^{th}$ non-mandatory share will have the following preamble, where each line is terminated by a single carriage return ("\n"):

- **Line 1.** Six space-separated integers: $i$, $L$, $M$, $N$, $C(N-1, M-1)$, and the size of a block in bits.
- **Line 1+i ($i < C(N-1, M-1)$).** Three space-separated integers: the segment number, the number of the first block it contains, and the number of the last block it contains.

If the share is a mandatory one, then the preamble is exactly the same except that it only contains two lines. Immediately following the preamble, a share contains each of its segments in exactly the order listed by the preamble, with no padding. If the second line of the preamble contained "2 105 108\n", for example, the data immediately following the preamble would be blocks 105 through 108, inclusive, of the original encrypted file, followed immediately by the next segment listed in the preamble.

**Joining the shares**. Given a number of shares, we first parse the preamble to ensure that all $L+C(N, M-1)$ segments are present in the provided shares and to find offsets within the shares to each segment. Joining then requires two passes: on the first pass, we scan through all blocks to obtain the key **K**, and on the second pass we use **K** and $\mathbf{T_0}$ to decrypt the file. Writing stops when the number of bytes specified in the file size block has been written out."BitSegment"

Closely related to the "BlockSegment" technique above, "BitSegment" assigns bits to shares according to the technique described below.

**Notation**. In the description of this algorithm, we standardize a number of definitions.

- A *segment* is an abstraction consisting of groups of bits. An entire file will be divided into segments for the splitting and joining processes. When there are $A$ total segments, then segment $i$ consists of bits $(i \mid A + i \mid (2*A) + i \mid (3*A) + i \ldots)$.
- [r] indicates the greatest integer not exceeding r
- C(n,k) denote the binomial coefficient "n choose k," the number of k-element subsets of a set of n elements, equal to $n!/(k!(n-k)!)$.
- Note also that all counting, whether of bits, segments, or shares, is zero-indexed.

**Assigning Segments.** A splittable file must contain at least C($N$+$L$, $M$-1) segments; any smaller number of segments will allow the file to be recovered with fewer than $L$+$M$ shares. The file to be split will be padded later with $D$ bits of meta-data, so if it contains fewer than $L$+C($N$, $M$-1)-$D$ bits, then it is padded at the end with randomly-generated bits until reaching this size. The prefix will have a block only the original file size stored in the big-endian order.

The bits are then divided up into segments as follows:

Let F be the number of bits in the given file, and define:

$$a = \left\lfloor \frac{F}{\binom{N-1}{M-1} \cdot \left( \frac{N}{N-M+1} + L \right)} \right\rfloor = \left\lfloor \frac{F}{\binom{N}{M-1} + \binom{N-1}{M-1} \cdot L} \right\rfloor, b = a \cdot \binom{N}{M-1}.$$

Let S = C($N$,$M$-1). For i=0 to C($N$,$M$-1) – 1, the $i^{th}$ segment () contains bits (i | S+i | 2S+i, …, aS+i). The remaining L mandatory shares are computed from the end of the file, by dividing the remaining (b*L) bits in a similar manner: mandatory share j contains bits ((a+1)S + j | (a+1)S + L + j | (a+1)S + 2L + j | … | (a+1)S + bL + j). These formulae will result in the final output shares being equally distributed in size.

**Assigning segments to shares**. To assign the C($N$, $M$-1) segments to the $N$ non-mandatory shares, we first enumerate all C($N$, $M$-1) of the ($M$-1)-element subsets of {0, 1, ..., N-1} in lexicographical order. We assign the remaining $L$ segments to the mandatory shares.

Bits are distributed sequentially: each chunk of C($N$, $M$-1) bits contains one bit belonging to each individual segment. These chunks are parsed sequentially one bit at a time, and each bit is written to the shares assigned to that segment.

**Writing out the shares**. The $i^{th}$ non-mandatory share will have the following preamble, where each line is terminated by a single carriage return ("\n"):

- **Line 1.** Six space-separated integers: $i$, $L$, $M$, $N$, C($N$-1, $M$-1).
- **Line 1+i (i < C($N$-1, $M$-1)).** Three space-separated integers: the segment number, the number of the first bit it contains, and the number of total bits it contains.

If the share is a mandatory one, then the preamble is exactly the same except that it only contains two lines. Immediately following the preamble, a share contains each of its segments in exactly the order listed by the preamble, with no padding.

**Joining the shares**. Given a number of shares, we first parse the preamble to ensure that all $L$+C($N$, $M$-1) segments are present in the provided shares and to find offsets within the shares to each segment. Writing stops when the number of bytes specified in the file size block has been written out.

## 6.3 "BitScatter"

The bit scattering algorithm (refer to Figures 2 and 3) is used to separate a stream of input bytes into separate output streams by bit. As such:

1. We first require a plain text *P*, a master key *K* and a session key K.
2. Perform an "m of n" key expansion (see Figure 2 to create $SK_0...SK_M$, which is equal in length to *P* * 8). We allocate arrays $S_0...S_M$ of the maximum length from all the $SK_0...SK_M$.
3. We iterate through each *SK* one integer at a time and read *y*. *y* represents the original position of the bit from *P*.
4. We read *P* at bit *y* and append the bit to the current share *S* which we are iterating through.

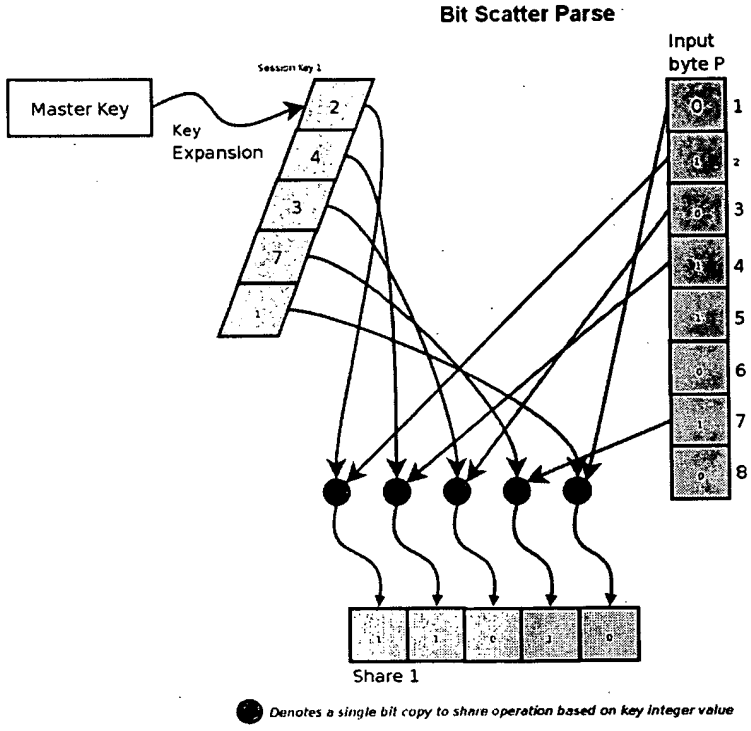



**Figure 2: Bit Scatter Parse**

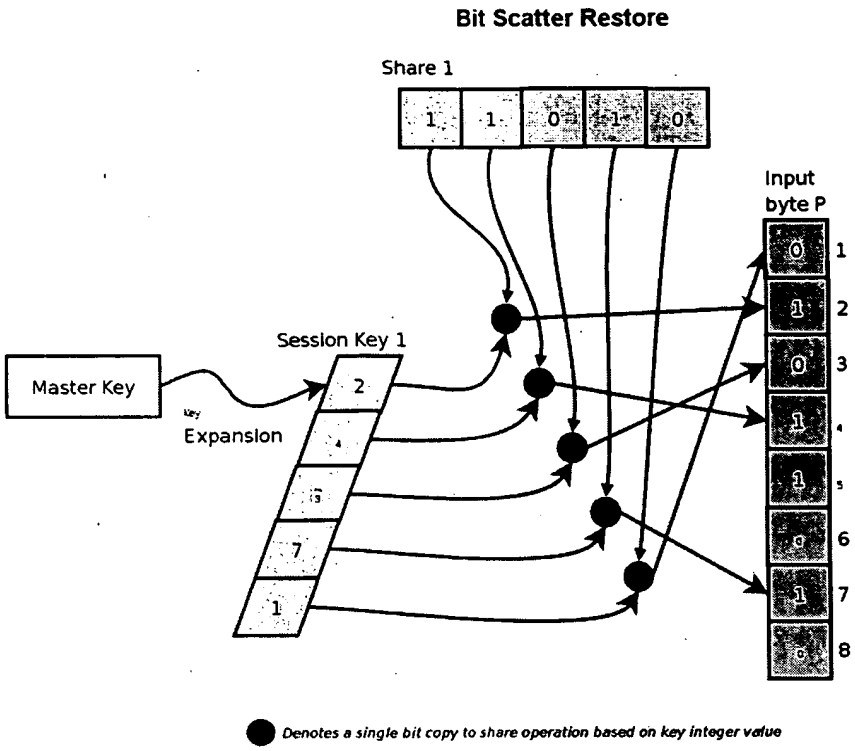



**Figure 3: Bit Scatter Restore**

### 6.3.1 M of N Key Expansion

The parser currently utilizes two external algorithms for key expansion, ANSI X9.82 (Draft July 2004) as a PRNG and an RC4 type algorithm for randomizing the order. Other key expansion algorithms may be integrated as external calls from the SecureParser.

An initial master key *MK* seeds the X9.82 running in AES-OFB mode, which produces *k*. The returned *k* blocks shall be of length *len* + 4 of plain text *P*, padding for AES.

$K_{len} = P_{len} += 4$

$k = aes\_ofb(\ MK,\ aes\_pad\_length\ (K_{len})\ )$

$len = aes\_pad\_length(\ P_{len}\ )$

We instantiate an array of integers *S* which represent the order of the bits in the plain text up to $2^{16}$. The array is initialized from $S_0$, $S_1$, ..., $S_{len}$. We also use a counters *i, j, and x.* The key len *k* is assumed to be equal to the array length of *S*. We then perform:

```
i = j = x = 0;
for( i ... len )
     j = ( j + kx + Si ) mod len
     swap( S[i], S[j] )
          x++;
```

This effectively randomizes the order of S, which we will use as a seed for our session keys $SK_0..SK_m$. Assuming we want to have a minimum of *n* shares out of *m*, we calculate

the percentage of key material from *S* which is required to reproduce *P*. Illustration 1 below shows an example of a 2 of 4 key split.

```
ratio = 1 - (1 / m) * (n-1)
```

Now we copy the *S* material using the ratio as the length to create our $SK_0 \ldots Sk_m$. Each successive copy starts at the byte following the end of the last byte on the S key, looping back to the beginning at the end of the key array. Illustration 1 below shows an example of a 2 of 4 key split.

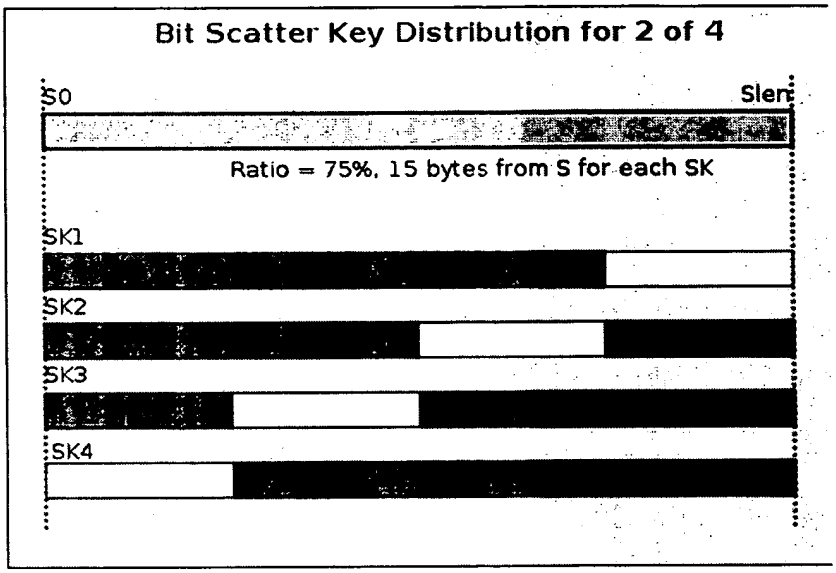



Illustration 1: Only the areas in red are copied from the yellow S key.

This ensures that the key material can correctly reproduce the plain text P. Finally, we iterate through each key re-randomize the SK with a random nonce:

```
for( SK0 ... Skm )
   i = j = x = 0;
   for( i ... len )
        j = ( j + nx + Si ) mod len
        swap( S[i], S[j] )
              x = (x+1) mod keylen;
```

## 6.4 "BitSplit"

The bit splitting algorithm is used to partition a single input stream byte of data into separate output streams at the bit level. Figure 4 below is provided to aid in understanding the BitSplit process.

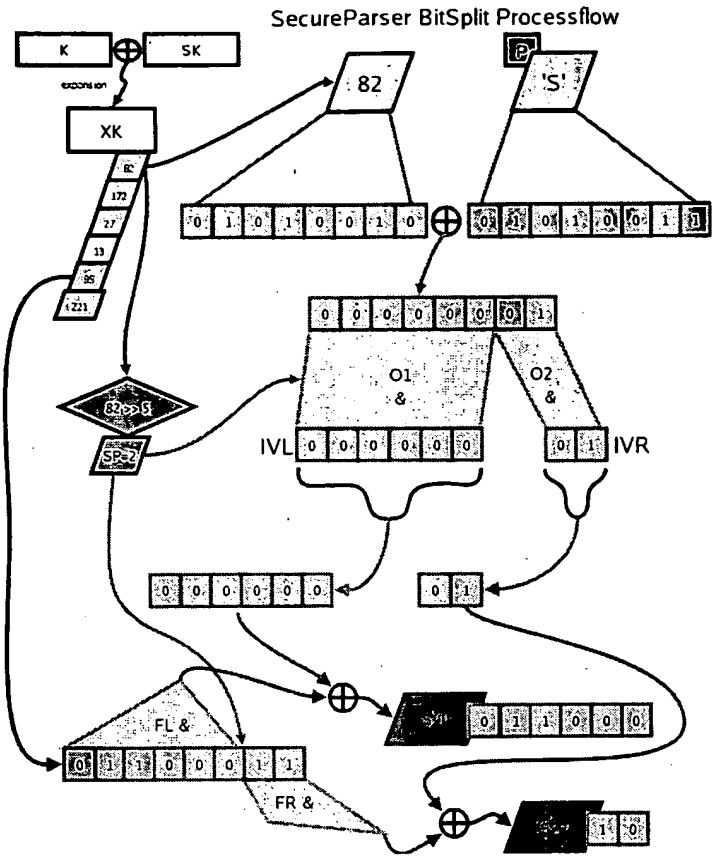



**Figure 4: Secure Parser BitSplit Process flow**

To split the data, the following takes place. We first require an input stream *P*, a optional Master Key *K* and a random session key SK:

1. The session key SK is XOR'ed with master session key K to create a session key XK.
2. The XK key is N of M key expanded (see "BitScatter") using to a length of $P_{len}$.
3. We right shift by 5 the XK byte to produce a split point *SP*.
4. We optionally XOR the *P* byte with the XK byte.
5. We create a mask $0_l$, with the bits set to 1 from the right using the *SP*. For instance, if the split point were 2, then the mask M value would initially be 00000011. The mask is simply used to separate the plain text byte into separate portions.
6. We AND operate the mask *M* with the *P* byte, to create IVR the right output split.
   `IVR = P & M`
7. We look ahead in the master session key 4 bytes in XK as a key feedback *F* and we AND operate the mask *M* with it to create the feedback value *FR*.
   `FR = F & M`
8. Next, we XOR the mask M with 11111111 to invert the bits (in our example, our output would be 11111100)
   `M = M XOR 1111111111`
9. We AND operate the mask *M* with *P*, to create IVL, the right output split.
   `IVL = P & M`

10. We AND operate F with the mask M with it to create the feedback value FL.

```
FL = F & M
```

11. FL is XOR operated with IVL, and FR is XOR operated with IVR, with the results stored in *SVL* and *SVR* respectively.

```
SVL = FL XOR IVL
SVR = FR XOR IVR
```

12. The SVL and SVR chunks are appended to the shares determined by $Dk_i$ by ANDing each bit position which corresponds to an output share. For instance, DK contains 0011 would correspond to share 1 and 2, as bits 1 and 2 are set.

### 6.4.1 Key and Output Share Storage

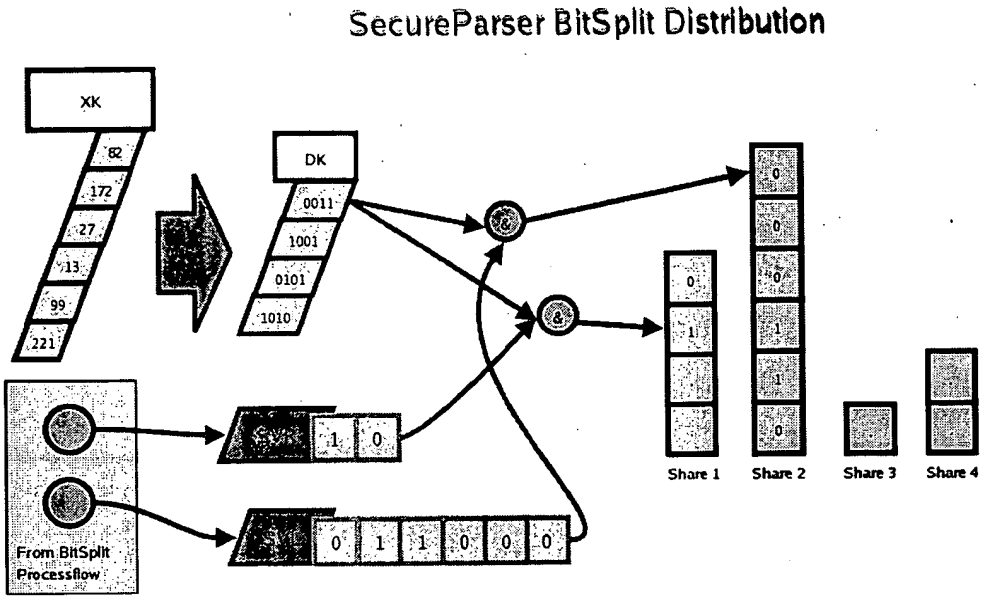



**Figure 5: SecureParser BitSplit Distribution**

Storing the shares (see Figure 5) is simply a matter of avoiding keys which were matched with shares, or dividing the key into multiple pieces across numerous shares. For a 4 of 4 restore fileset, the following are output from the SecureParser.
Session Key 1: Output share 1, 2, 3, & 4 .

These can be stored into 4 separate files: (This would be 4 of 4 splitting) First, the session key is divided into 4 parts. 1a, 1b, 1c, and 1d. Then we store them into 4 files, along with the share output generated by the SecureParser.

| | | |
|---|---|---|
| 1 | 1d | Share 1 |
| 2 | 1c | Share 2 |

| | | |
|---|---|---|
| 3 | 1b | Share 3 |
| 4 | 1a | Share 4 |

# 7 Conclusion

This document outlines an architecture and proposed algorithms for efficient data splitting with strong security guarantees. These solutions are among several approaches to the problem of securely splitting data. The algorithms enclosed within this document do not preclude the development of additional algorithms and security features in future versions of the SecureParser.

# **Security First** - Overview

- SecureParser provides a multiplicative improvement to existing security and disaster recovery
  - This is achieved by the breaking up of, or parsing, any piece of data into some number of shares
    - Simply, the SecureParser cryptographically splits any piece of data at the bit level into a defined number of shares
      - Shares can then be geographically and physically dispersed
      - Each share can be layered with additional encryption algorithms at multiple points in the splitting process for added security



EX1015, Page 293

# **Security First** - Overview (cont'd)

- This simple concept solves two seemingly divergent problems
  - Attacking individual shares does not provide enough information to reconstruct data
  - By providing redundant information in shares, fault tolerance can be achieved



EX1015, Page 294

# **Security First** - Overview (cont'd)

- SecureParser was initially designed to protect large scale storage of biometric templates and has since evolved into the general purpose of protecting all data at rest and in motion



EX1015, Page 295

# **Security First** - Overview (cont'd)

- Market Validation
  - Customer installations and pilots with government agencies and groups that include: USSOCOM, DIA, and US Air Force
    - Government agencies further validate the high level security and operational flexibility achieved
  - Corporate partners include EMC, Unisys, and Electronic Warfare Associates (EWA)
  - Others working, Yotta Yotta, IBM, Brocade, Blue Arc

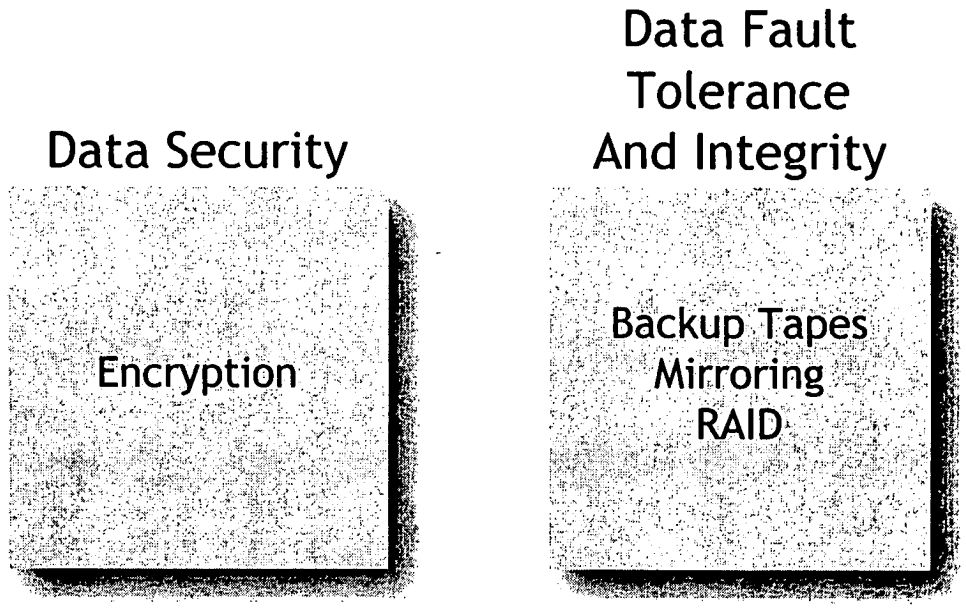


EX1015, Page 296

2/4

# **Security First** - Two Industry Primary Functions

**The traditional view of data redundancy/ fault tolerance and data security**




*These are separate problems and processes*

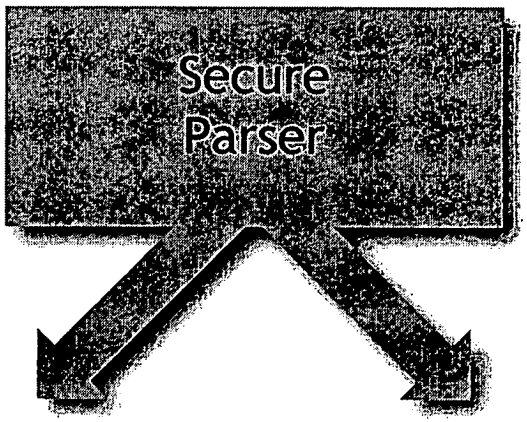


EX1015, Page 297

2/15

# **Security First** - SecureParser Overview

- The SecureParser provides two critical and configurable solutions with one process: Fault Tolerance and Security




Fault Tolerance Security



EX1015, Page 298

# **Security First** - SecureParser Overview (cont'd)

- Physical, cryptographic separation is the differentiator
  - Randomized, cryptographic parsing of data into unique shares
  - Physical separation of shares
    - Partitions, Disks, Arrays, SANs…
  - Layered, configurable encryption of any type
  - Flexible key management
  - Fault tolerance for disaster recovery

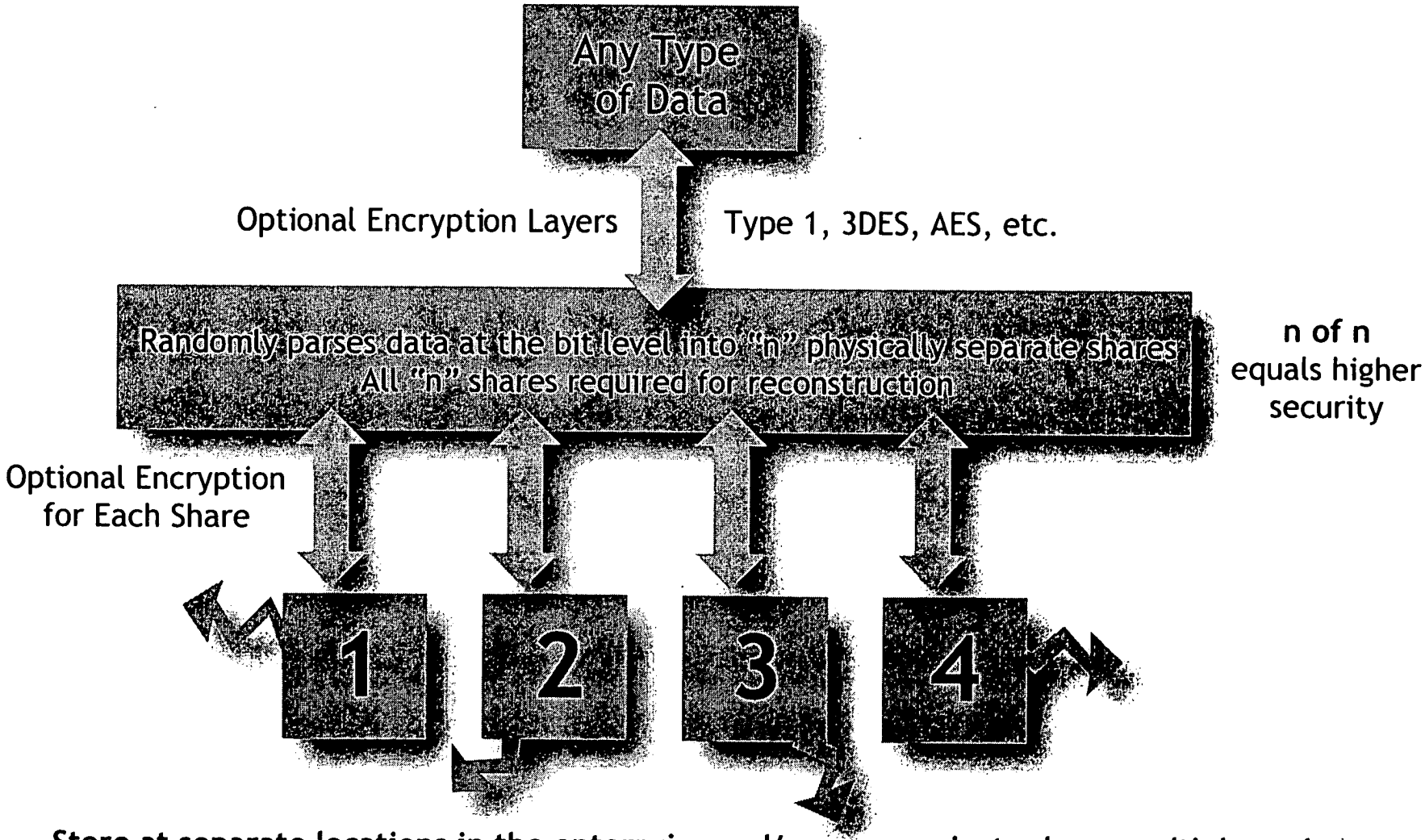
*When one or more shares are physically distributed within a network, a security breach is effectively removed*



EX1015, Page 299

3917

# Security First - The Parsing Process



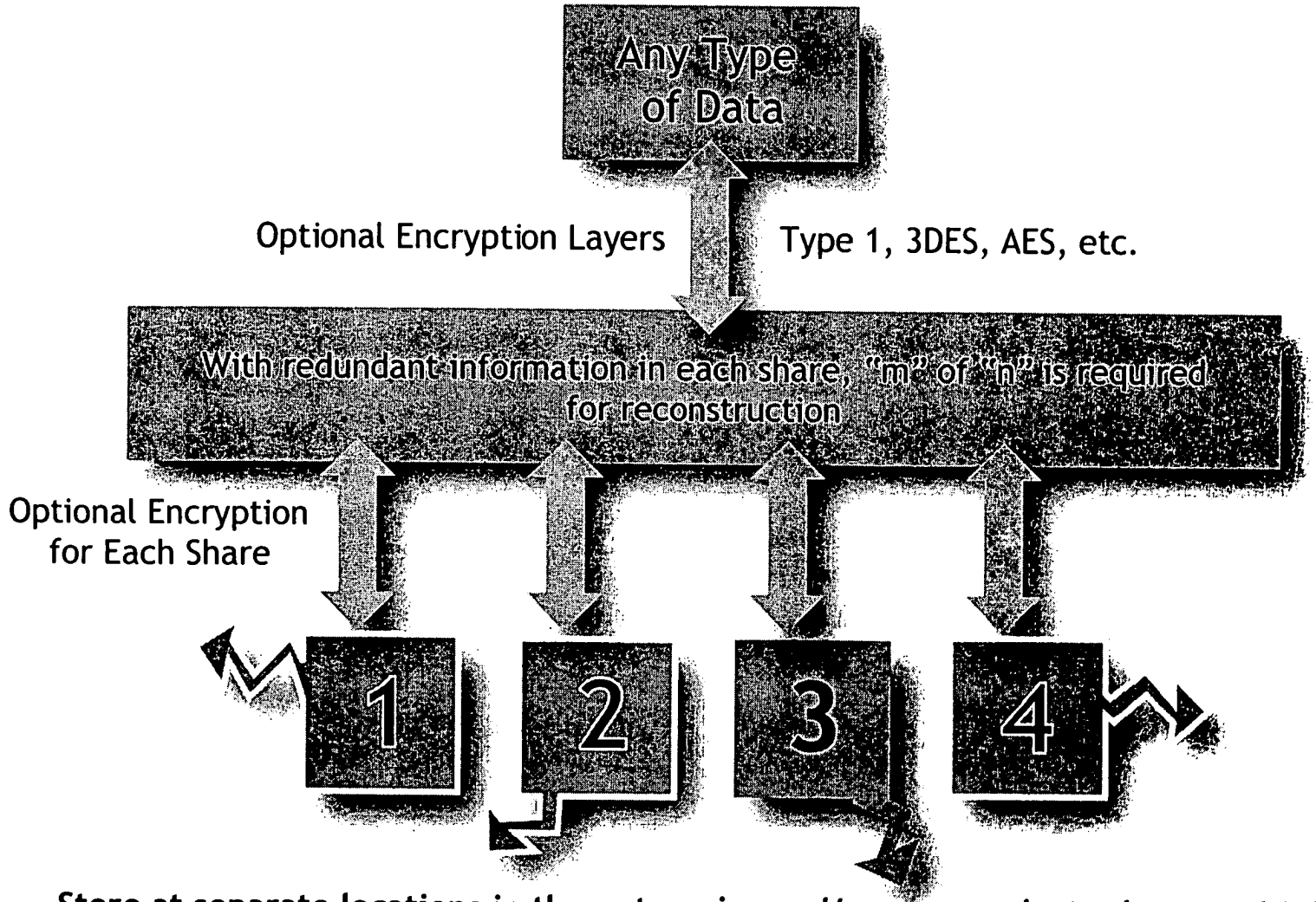

**Store at separate locations in the enterprise and/or communicate down multiple paths**



EX1015, Page 300

248

# Security First - The Parsing Process




**n of n** equals higher security

**m of n** equals fault tolerance and redundancy

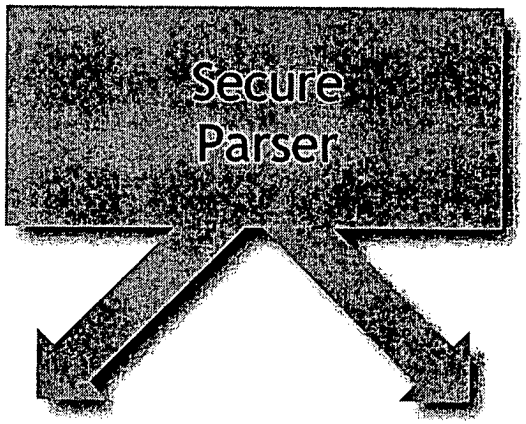
**Store at separate locations in the enterprise and/or communicate down multiple paths**



EX1015, Page 301

# **Security First** - The Parsing Process (cont'd)

**The SecureParser provides two critical and configurable elements with one process:**
**Fault Tolerance and Security**




Fault Tolerance

Only "m" of "n" shares are needed to survive in order to successfully reconstruct data

Security

Attack requires access to "m" shares and keys to compromise data

*Pre- or post-parsing, layered encryption exponentially complicates an attacker's task*

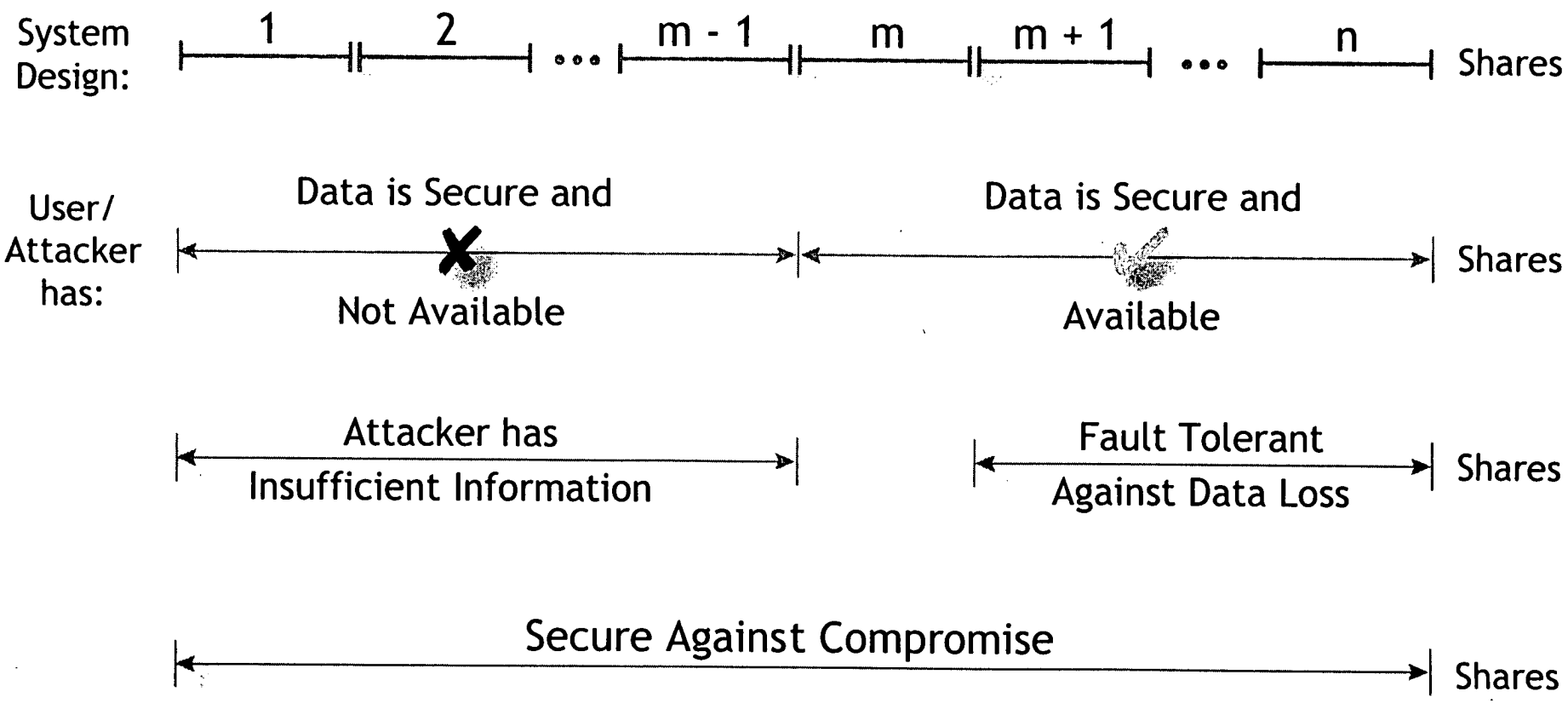


EX1015, Page 302

300

# Security First - Parser Integrated Key Management

**The following diagram illustrates data protection using the SecureParser integrated key management feature:**

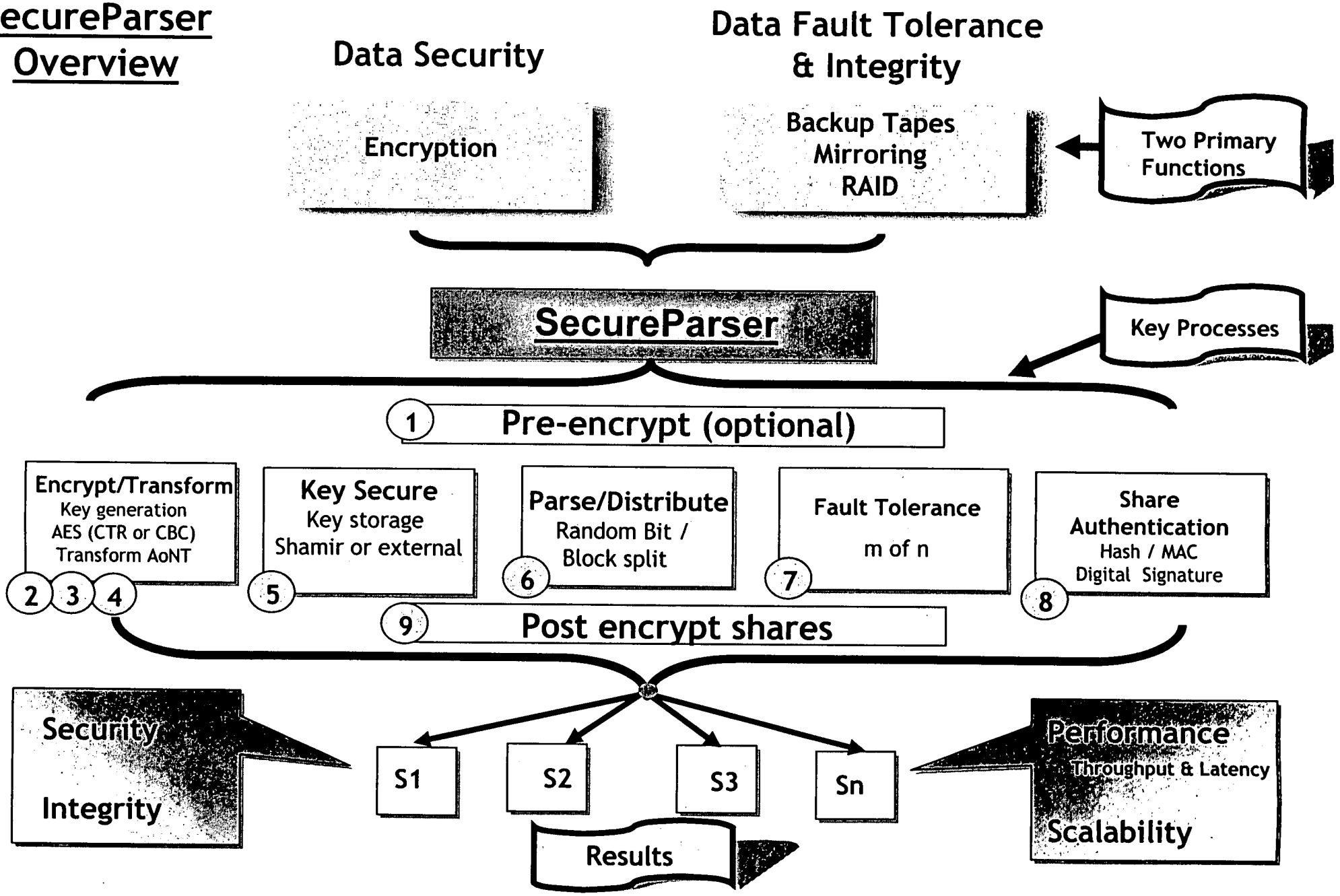





EX1015, Page 303

301

# SecureParser Overview






EX1015, Page 304

# SecureParser - 9 Step Process

1. **Pre-encryption (optional):** The plain text is optionally encrypted using AES under a 128, 192 or 256-bit key. This supports FIPS Compliance and compatibility with sites using external key management systems.

2. **Encryption Split Session Key Generation:** A session key is generated using a cryptographically secure pseudo-random number generator algorithm (CSPRNG). This key is unique for each session and may be 128, 192 or 256 bits in length.

3. **Optionally Pre-encrypted:** data is encrypted in its entirety using the AES encryption algorithm and under the split session key. Other encryption algorithms can be integrated to support government or commercial encryption requirements.

4. **Transform:** The split session encryption key is processed using the Package Transform, or a form of "All or Nothing Transform (AoNT)". This procedure prevents key exposure when fewer than the required shares are exposed.

5. **Key Secure:** The above transform key or split session key is securely divided into key-shares using a "split knowledge" key splitting technique such as the Shamir algorithm. Each key-share is distributed to one data share.

6. **Parse / Distribute:** The data is partitioned into segments using one of several distribution algorithms. The data may be distributed to shares randomly at the bit or block level.

7. **Fault Tolerance:** When fault tolerant mode is used, additional information is written to each share to allow for restoration of the data from less than the total number of shares. This method is referred to as "m of n"

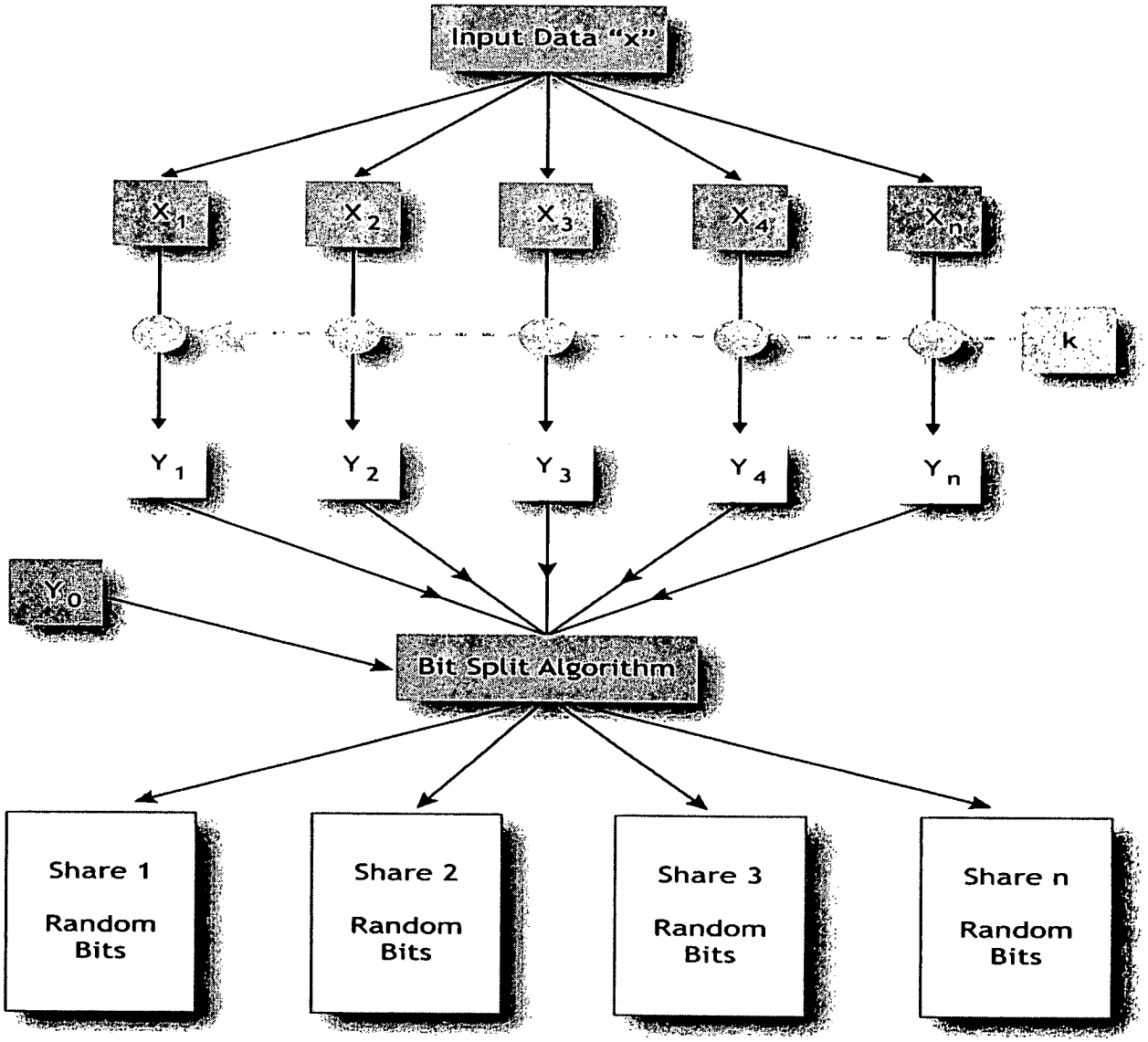
8. **Integrity:** Integrity information is written to each share in the form of a hash value, which allows for the detection of damaged or malformed shares. For additional protection, the shares may also be protected by use of a digital signature or Message Authentication Code (MAC).

9. **Post-encryption (optional):** The resulting shares may optionally be encrypted under a key provided by the user.



EX1015, Page 305

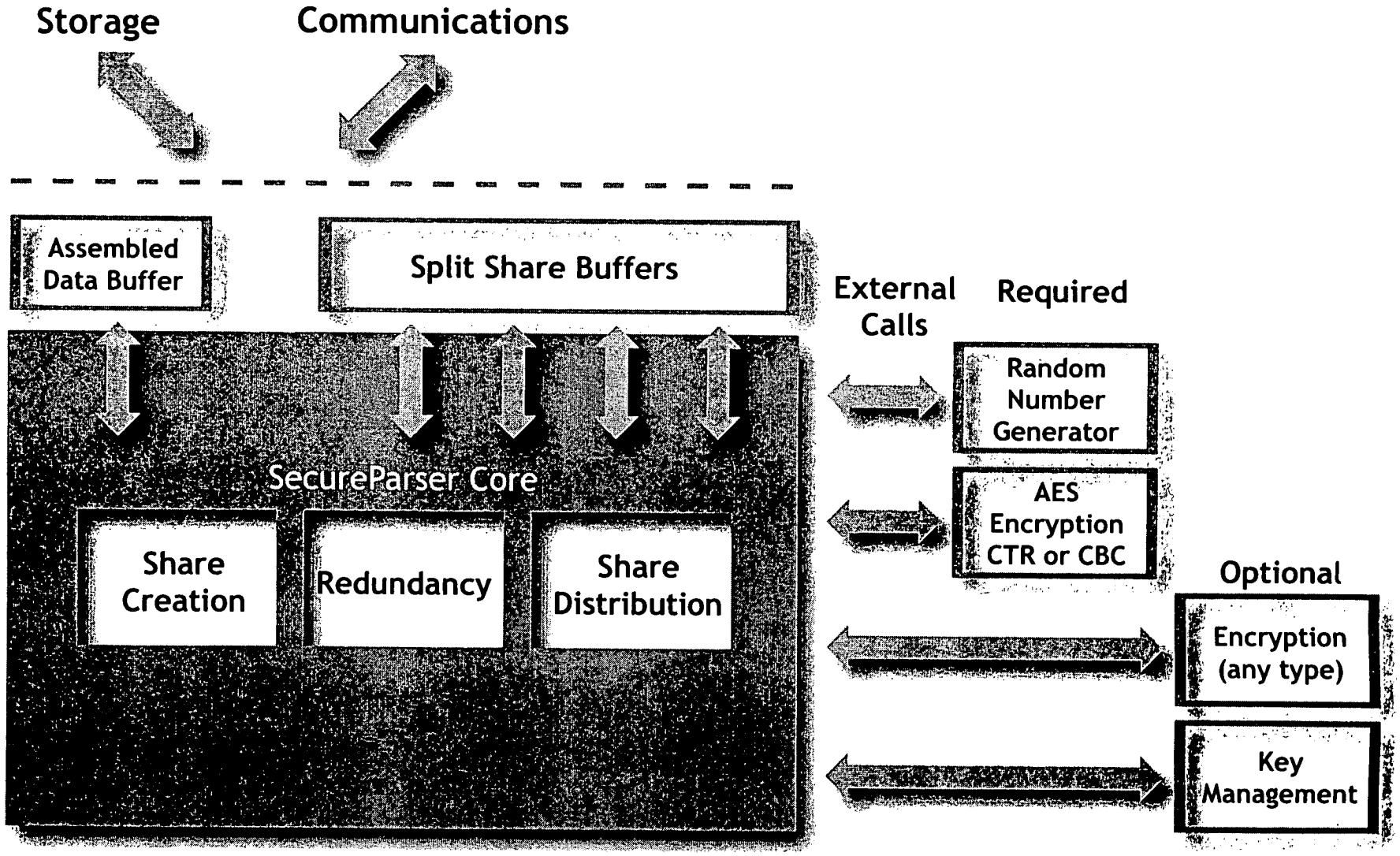
# Security First - SecureParser - Functional Flow






EX1015, Page 306

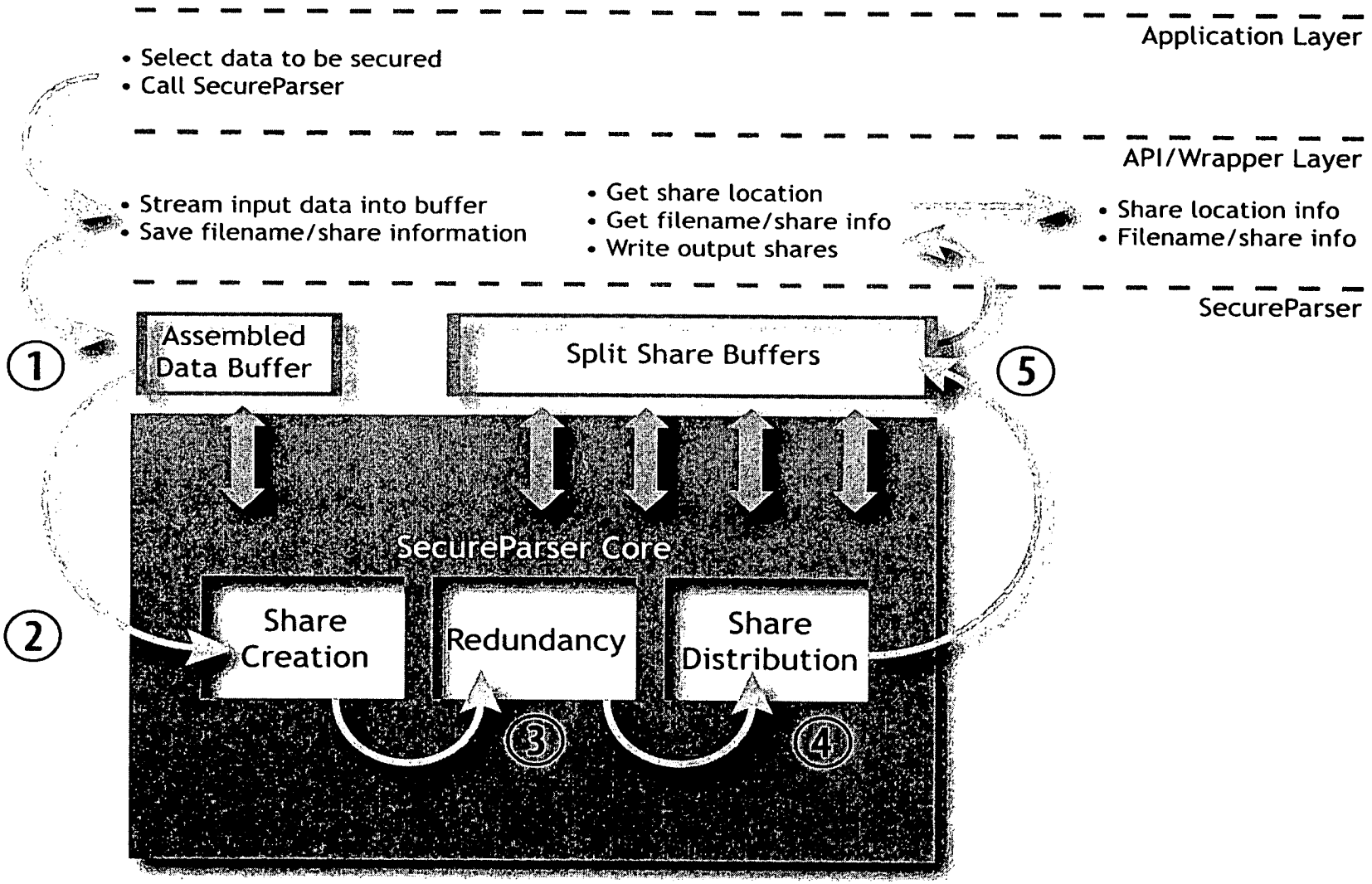
# Security First - SecureParser Toolkit at Work






EX1015, Page 307

# Security First - SecureParser- Integration






EX1015, Page 308

# SecureParser -Toolkit Combination Options

- The SecureParser Toolkit has over 1,200 combinations of options to choose from when determining what type of data Transformation, Share Distribution scheme, Fault Tolerance Mode, Key Secure and Share Authentication is desired.
  - 123 combinations using the AoNT transform & AES CTR Mode
  - 123 combinations using 3DES Encryption
  - 123 combinations using AES counter mode
  - 123 combinations for each other encryption one chooses
  - 369 combinations if choose to not Transform data
    - Not shown in these charts but available.
  - 246 combinations based on various key lengths
  - Ability to use just about any other requested encryption
- All options are recommended to be done based on the threat and Risk Scenario attempting to secure data from

- Additional other configurations are possible such as random number generation, post encrypting of shares etc.



EX1015, Page 309

367

# SecureParser ToolKit Options (cont'd)

| Options | Transform | Distribution | | | Fault Tolerance | | | Key Secure | | | Share Authentication | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AONT | Bit Split | Block Shuffle | Ordered Block | M of N | M of N | Layer | Shamir | Other | Hormel | None | Hash | MAC | Dig Sig |
| 1 | X | | | | | | | X | | | | | | |
| 2 | X | | | | | | | | X | | | | | |
| 3 | X | | | | | | | | | X | | | | |
| 4 | X | X | | | X | | | X | | | X | | | |
| 5 | X | X | | | X | | | X | | | | X | | |
| 6 | X | X | | | X | | | X | | | | X | | X |
| 7 | X | X | | | X | | | X | | | | | X | |
| 8 | X | X | | | X | | | X | | | | | X | X |
| 9 | X | X | | | X | | | | X | | X | | | |
| 10 | X | X | | | X | | | | X | | | X | | |
| 11 | X | X | | | X | | | | X | | | X | | X |
| 12 | X | X | | | X | | | | X | | | | X | |
| 13 | X | X | | | X | | | | X | | | | X | X |
| 14 | X | X | | | X | | | | | X | X | | | |
| 15 | X | X | | | X | | | | | X | | X | | |
| 16 | X | X | | | X | | | | | X | | X | | X |



EX1015, Page 310

# **Security First** - Where SecureParser Fits

- Overall the SecureParser can be applied wherever the greatest benefit is to be realized
- Provides complete flexibility and the choice of integration location within the system stack



EX1015, Page 311

309

# **Security First** - Where SecureParser Fits (cont'd)

**The following demonstrates some general areas where the SecureParser can reside:**

- Application servers
- Host bus adapters
- Storage fabric switches and directors
- Storage arrays
- Tape devices
- Virtualization abstraction layers
- Remote mirroring/replication storage applications
- Routing and Communications
- Backup and recovery software
- Storage management software and interfaces
- Remote storage management access points
- Notebook computers
- Storage network management software
- Storage MAN/WAN connectivity



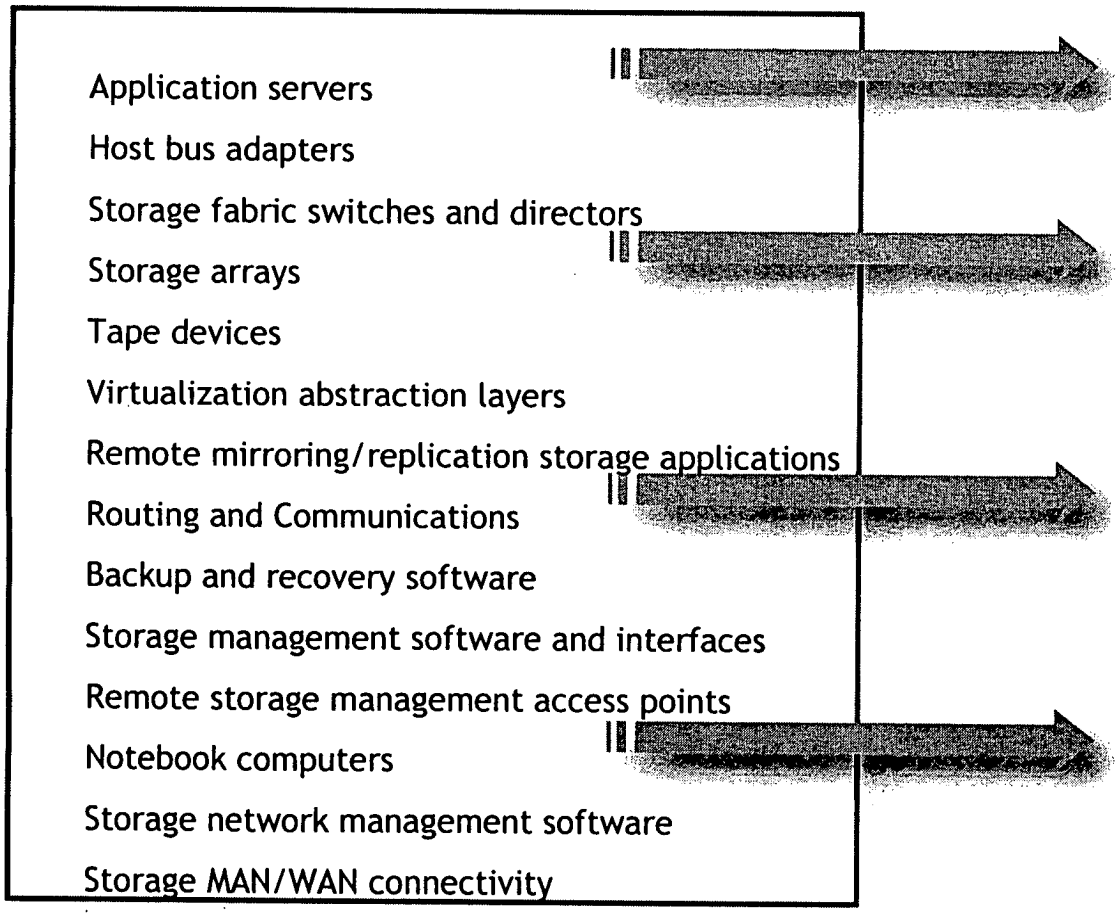
- Database Companies
  - Goal: Hardware independence
  - Where: In application
- Storage Companies
  - Goal: Disk and application independence
  - Where: In SAN controller
- Switches, Routers
  - Goal: Hardware and software independence for secure communications
  - Where: In router IOS
- Chip Companies
  - Goal: Application independence
  - Where: Embed in hardware and USB controllers



EX1015, Page 312

# Security First - Where SecureParser Fits (cont'd)

**The following demonstrates some general areas where the SecureParser can reside:**




- **EMC**
  - Storage Solutions
    - Business Continuity
    - High Availability
    - Security
  - Government Operations
    - Multi -Level Security
    - Securing Communications
- **Yotta Yotta**
  - Integrated into OS
- **Blue Arc**
  - Embed into appliance
- **Brocade**
  - Integrated into Network Switch



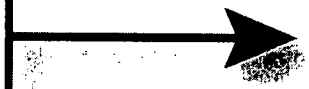
EX1015, Page 313

3/11

# **Security First** - Where SecureParser Fits (cont'd)

**Many possible applications within All State and below are just a few examples:**

- Application servers
- Host bus adapters
- Storage fabric switches and directors
- Storage arrays
- Tape devices
- Virtualization abstraction layers
- Remote mirroring/replication storage applications
- Routing and Communications
- Backup and recovery software
- Storage management software and interfaces
- Remote storage management access points
- Notebook computers
- Storage network management software
- Storage MAN/WAN connectivity



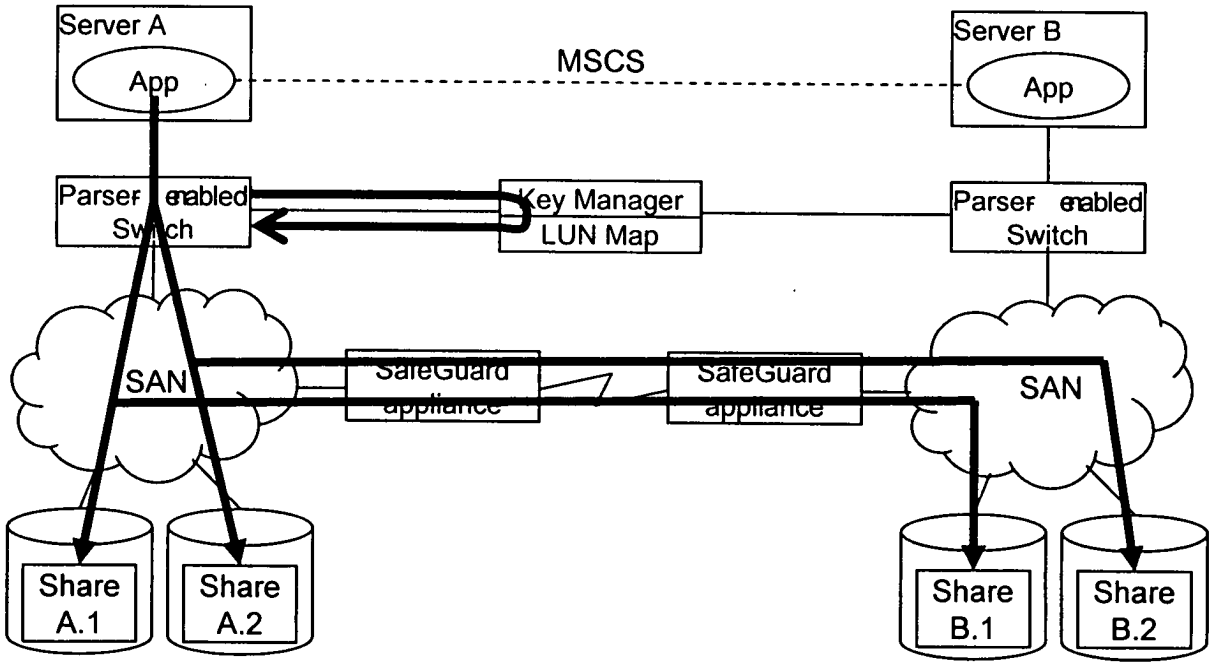
- **Unisys**
  - Parser enabled Switch
  - Parser enabled HBA
  - Others



EX1015, Page 314

# SecureParser/SafeGuard Disaster Recovery.

- Parser-enabled switch



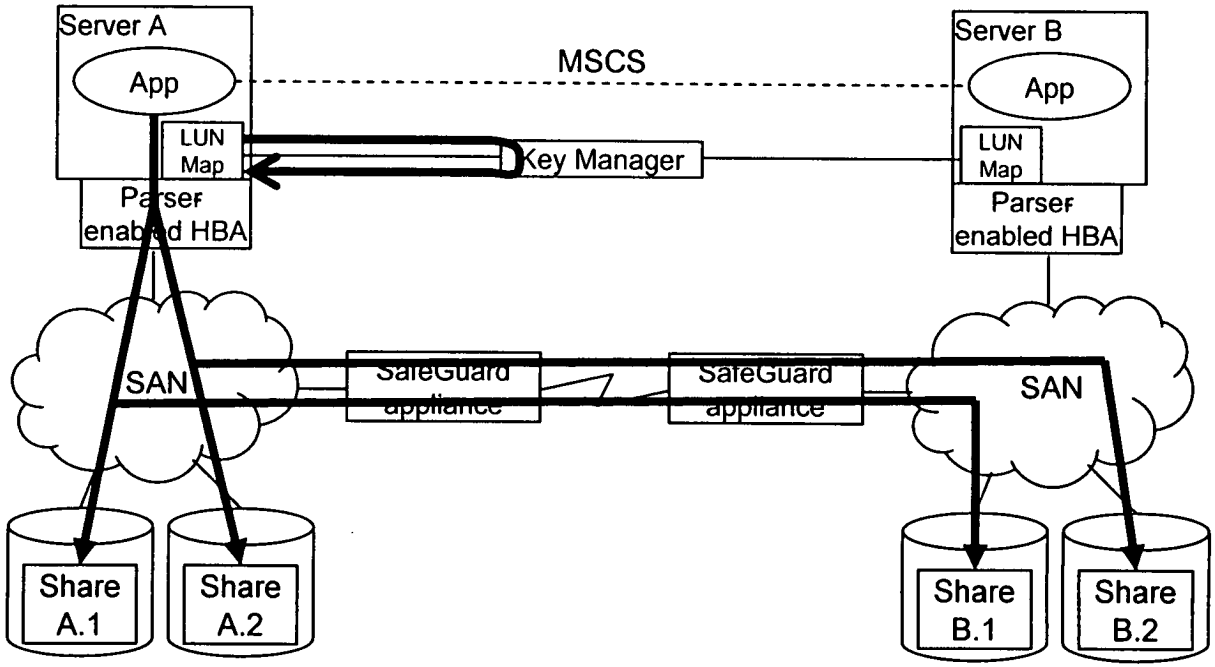

1) App writes data to LUN X
2) Parser enabled switch looks up key and LUN mapping
3) Parser enabled switch parses data to shares A.1 and A.2
4) SafeGuard captures the writes to A.1 and A.2 and replicates them remotely to B.1 and B.2



EX1015, Page 315

# SecureParser/SafeGuard Disaster Recovery.

- Parser-enabled HBA




1) App writes data to LUN X
2) Parser enabled HBA looks up key and LUN mapping
3) Parser enabled HBA parses data to shares A.1 and A.2
4) SafeGuard captures the writes to A.1 and A.2 and replicates them remotely to B.1 and B.2



EX1015, Page 316

# Security First - SecureParser Value Proposition

- **Performance**
  - Real and measurable software performance enhancements
- **Data Integrity**
  - Overall improvement of data integrity through fault tolerant mode and real-time disaster recovery
- **Scalability**
  - SecureParser toolkit works with any hardware platform and software application
- **Latency**
  - Minimizes latency by providing faster access to data
- **Security**
  - Offers the ultimate in security through physical separation of data at the bit level, while working with any type of encryption



EX1015, Page 317

# **Security First** - SecureParser and YOTTA YOTTA

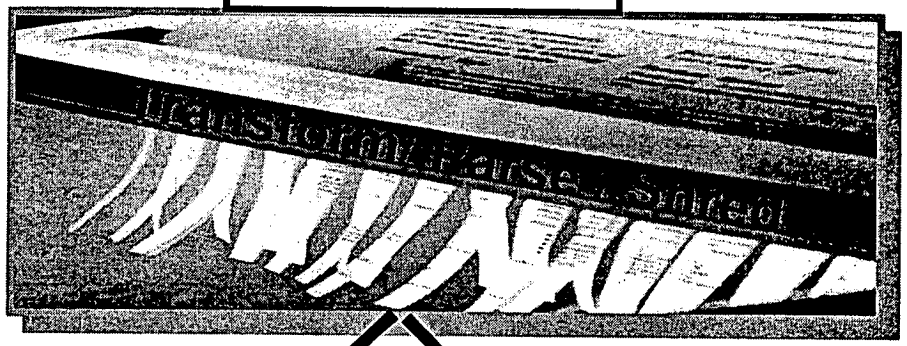
- **You get High Performance Access to Secure and Robust Disaster Recovery and Fault Tolerant Data**
- **Integration at OS level (MOS)**
- **SecureParser throughput expected 30+ GIGABYTES/SEC**
  - 20 Blades & 16Fiber Channels/blade @ 200Megabytes/Sec
    - 320 Fiber Channels
  - 1Blade=1.6 Gigabytes aggregate throughput
  - Plus four 4xIB (Infiniband) ports (1 Gigabyte/IB) + 2 Gigabyte Ethernet ports
- Nothing like it or close to it in the Market place



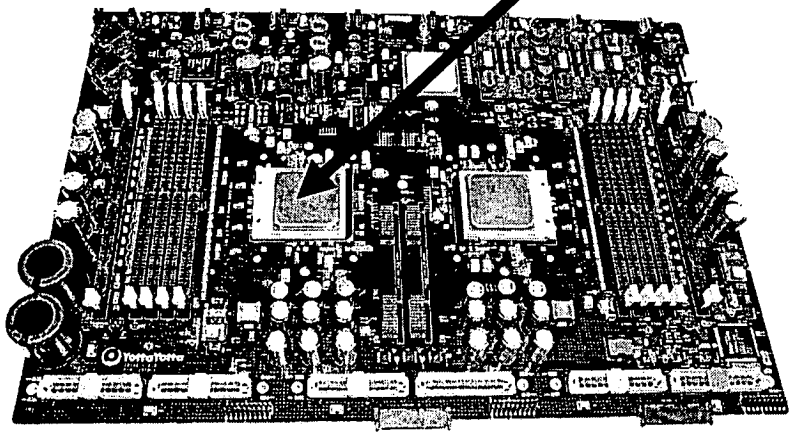
EX1015, Page 318

**Security First-** SecureParser integrated into YOTTA YOTTA GSX-3000 OS-(MOS)

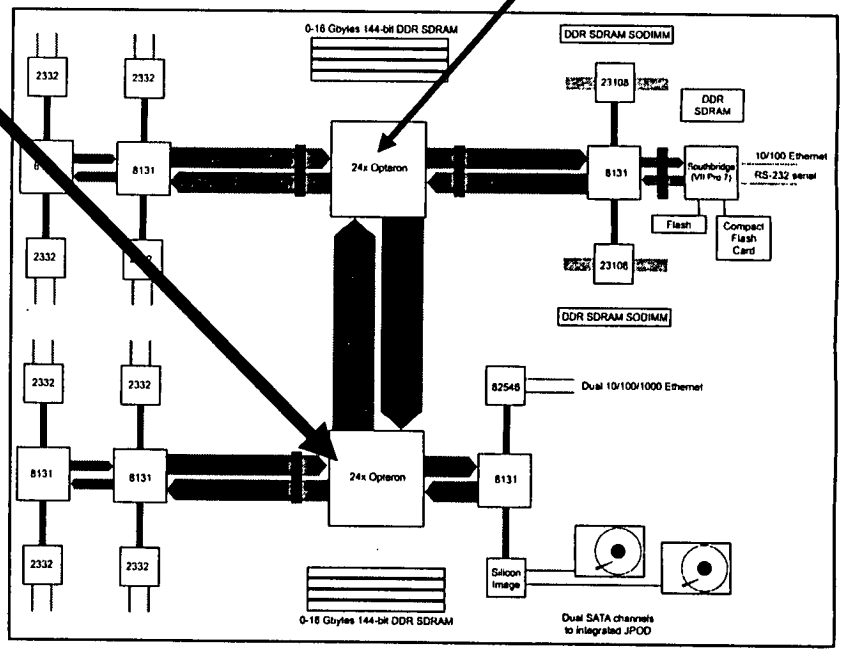
**SecureParser**



**Additional Applications**



**GSX-3000Trinity (Main Board))**

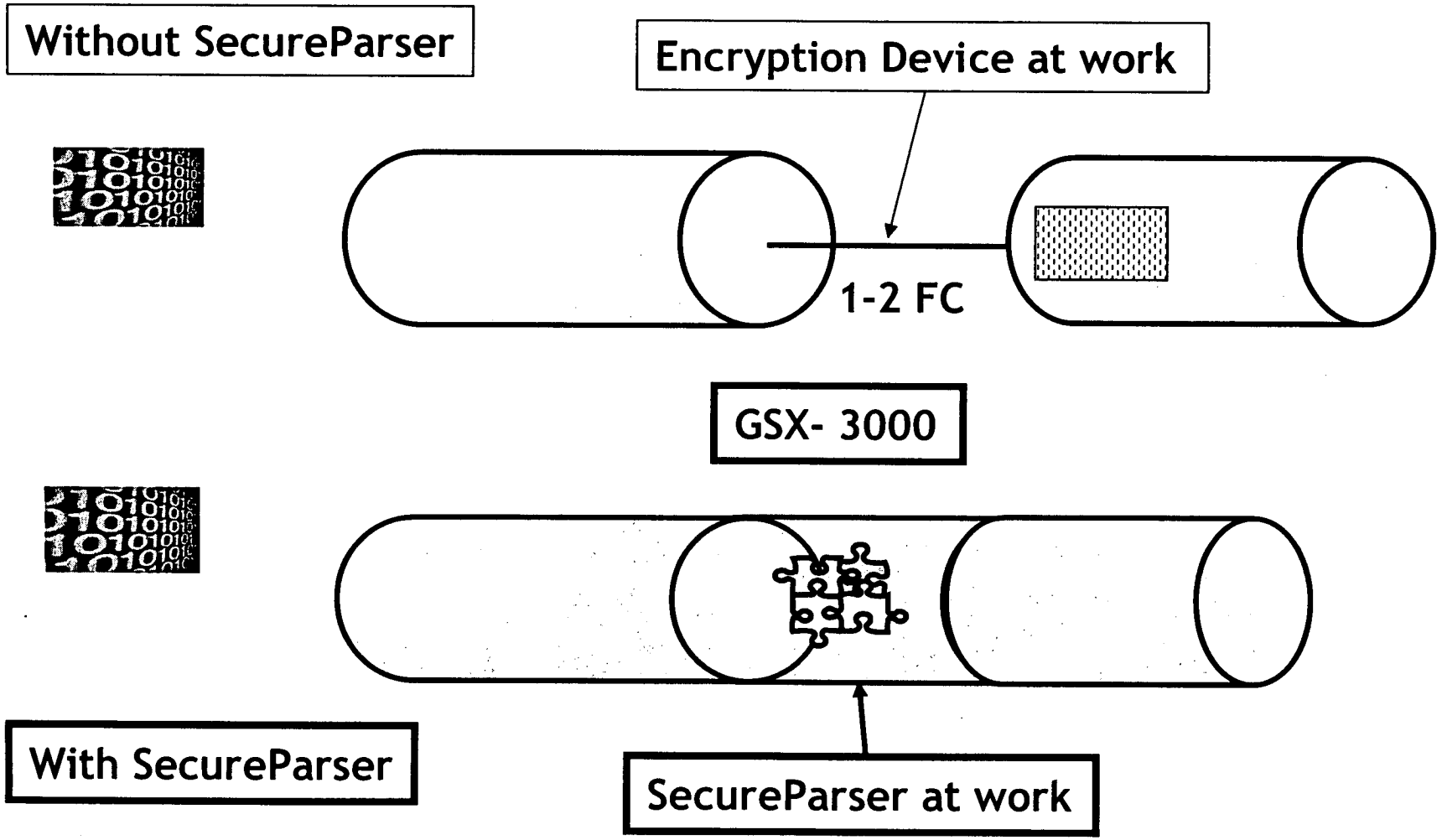



**GSX-3000 Node - Internal Topology**



EX1015, Page 319

# Security First - Securing data Without and With SecureParser






EX1015, Page 320

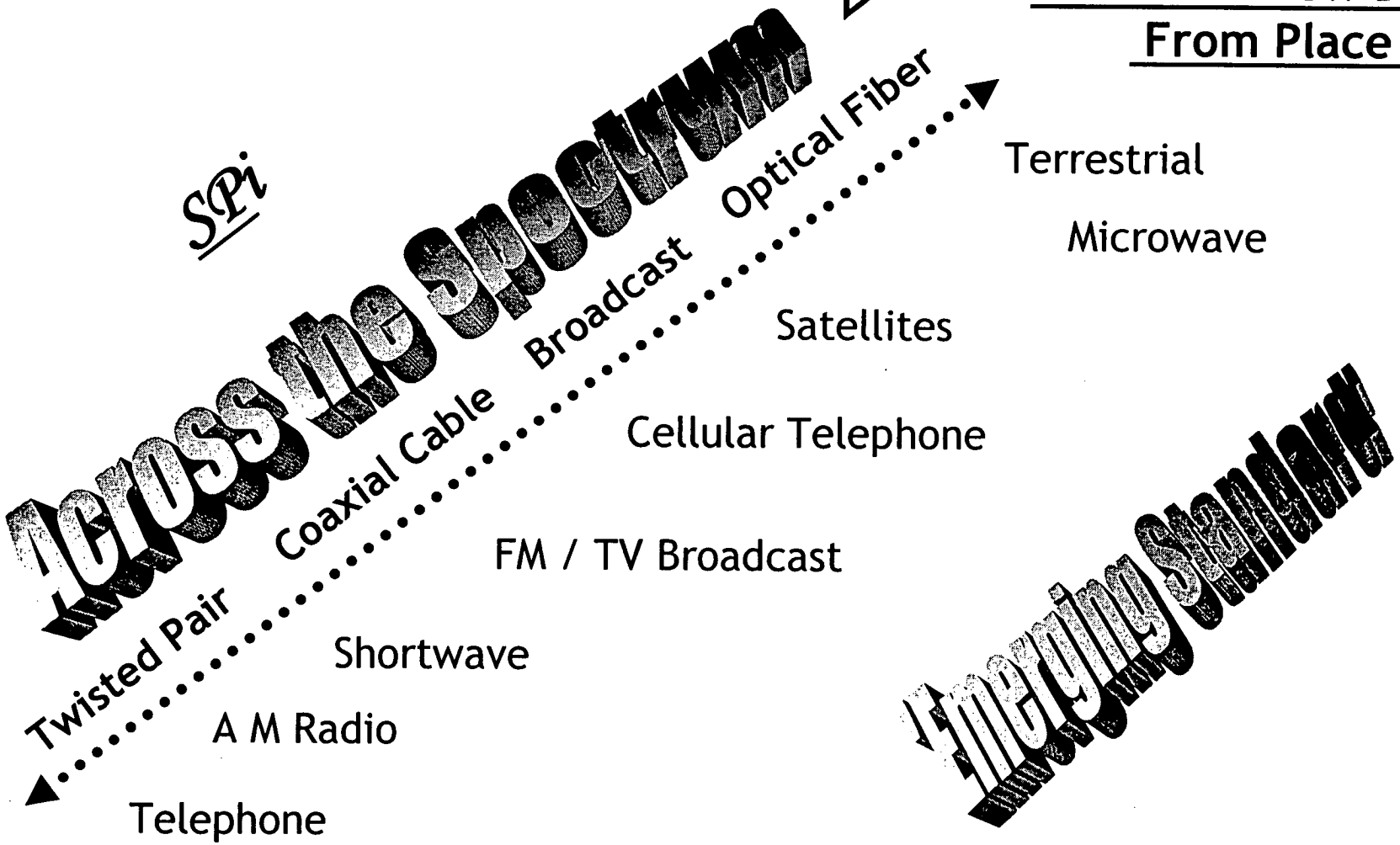
3/4

SPi

Fault Tolerant
Cross Platform
Net-Centric

# Securing 'Data in Motion'

**No Matter How Data Gets From Place to Place**



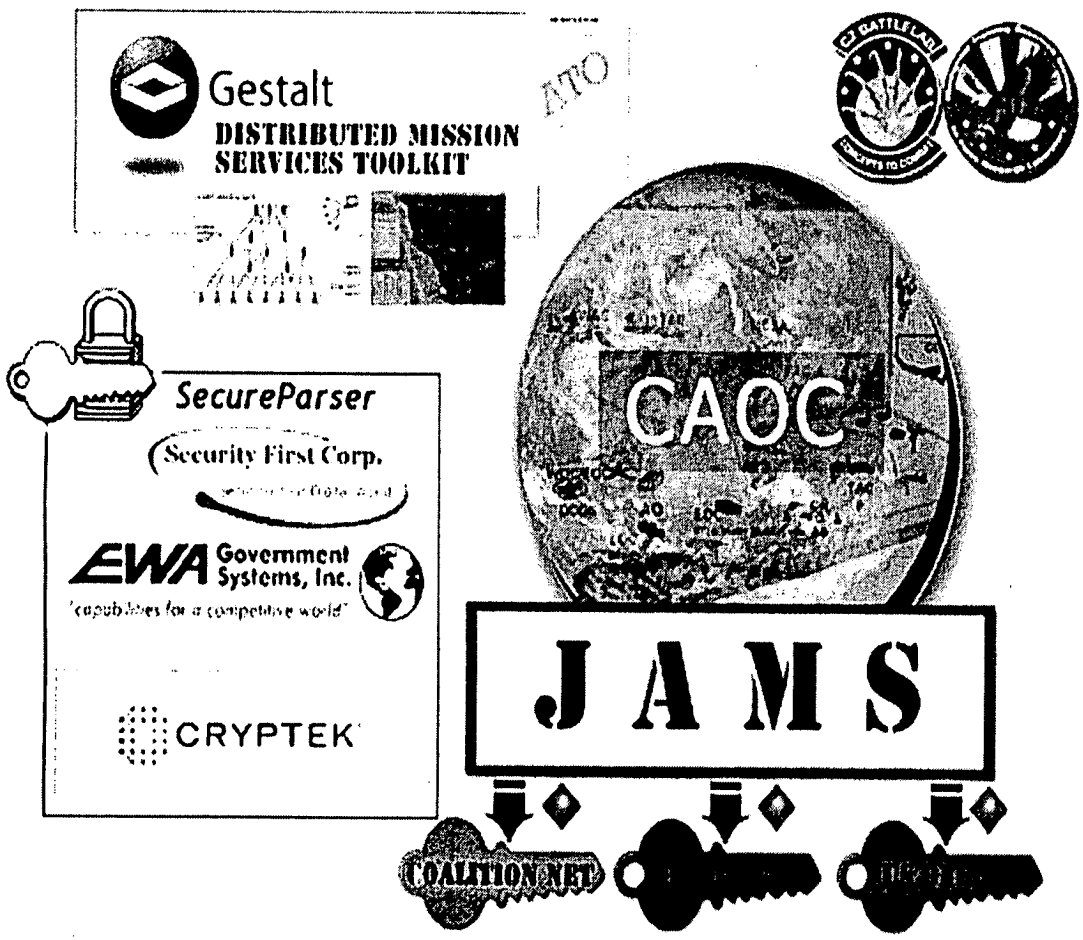



EX1015, Page 321

3/9

# Joint Air Mission Services (JAMS)

## *SPONSOR:* USAF Command & Control Battlelab (C2B) and Air Force Research Laboratory (AFRL)




***JAMS,*** is a net-centric information delivery system for the warfighter that allows web access to air mission data for improved collaboration and interoperability, while incorporating a multi-level secure, cross-domain solution for the Air Operations Center, using the *SecureParser*.

--To transform current Command & Control (C2) systems to allow data to be shared with coalition partners in a multi-level security, cross-domain environment is essential.

-- The ability to allow coalition partners to interact on a U.S. controlled network and yet effectively control what each coalition partner is authorized to see is our primary objective.

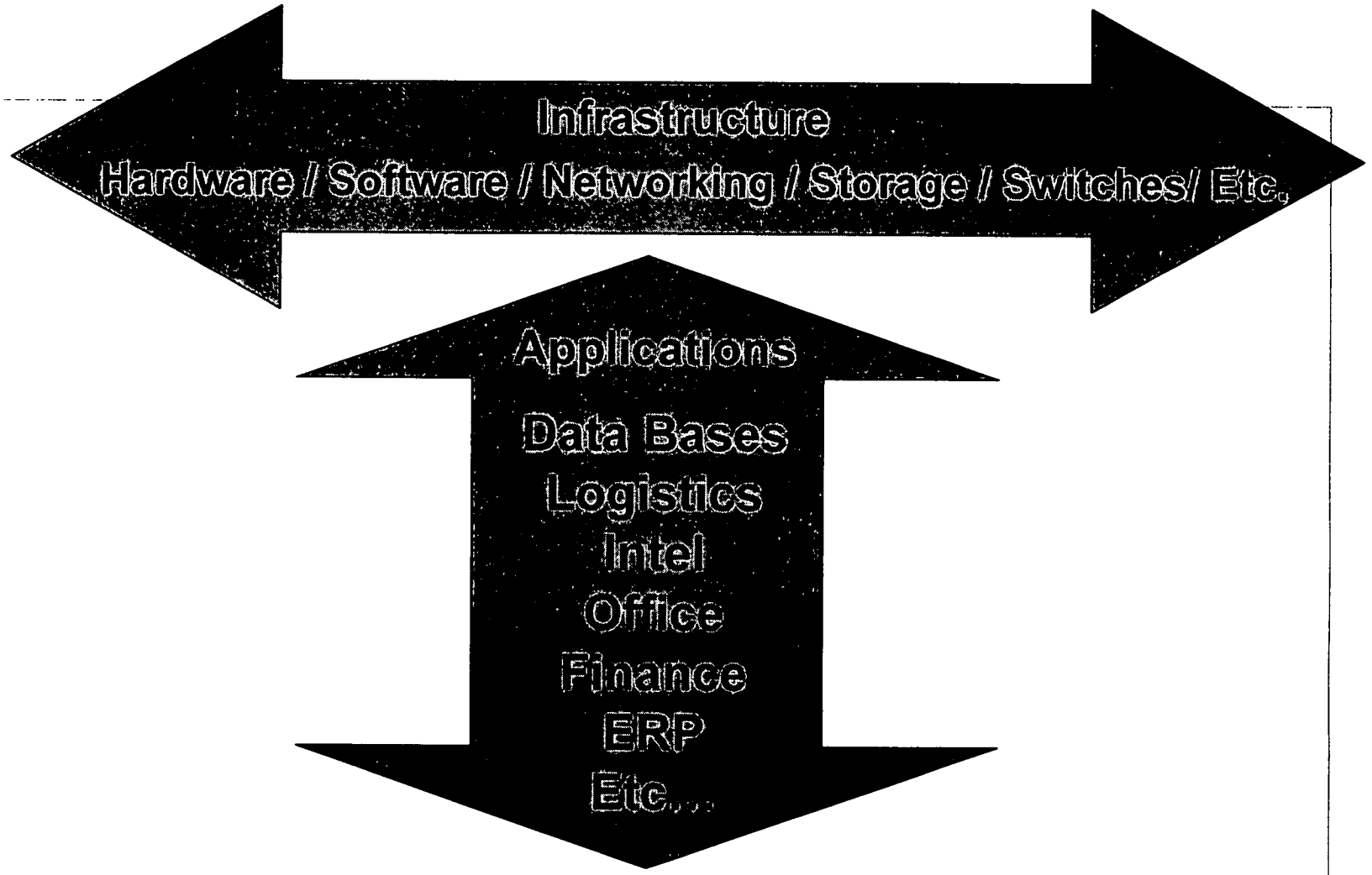
**JAMS** will demonstrate how Gestalt's web-based C2 applications can operate in a secure cross-domain environment (PL-4).

-- **JAMS** leverages other C2 systems of record and authoritative data sources, such as TBMCS. It will potentially support TBONE, an ongoing C2B initiative to re-engine the air battle planning & execution system for the Combined Air Operations Center (CAOC).



EX1015, Page 322

# Information Touch Points



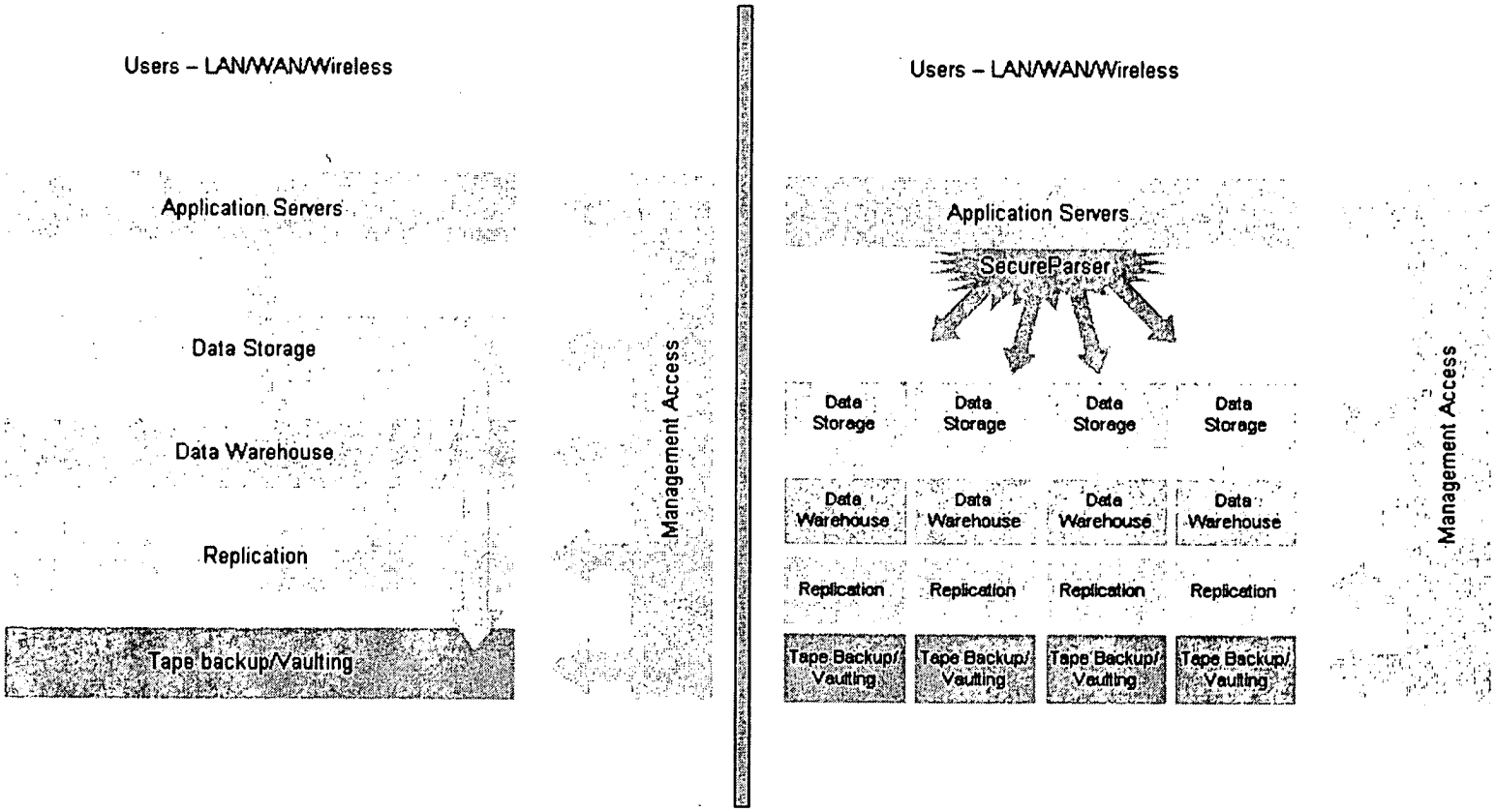

**Anywhere You Write or Read Information**



EX1015, Page 323

# Security First - SecureParser in Data Storage

**Typical Data Storage Environment** ⇨ **SecureParser Implementation**




***Implementing the SecureParser eliminates vulnerabilities and prevents access to any information in its entirety.***



EX1015, Page 324

326

# Security First - A Qualitative Look

EMC² where information lives

## Top Five Data Storage Area Network (SAN) Security Threats and Risks

| | Threat Type | Current Risk Level | With SecureParser |
|---|---|---|---|
| 1 | Management Interface | HIGH | LOW |
| 2 | Switch to switch | MEDIUM | LOW / NONE |
| 3 | Server to Storage Array | MEDIUM | LOW / NONE |
| 4 | Server or storage array to network connection | MEDIUM | LOW / NONE |
| 5 | Denial of Service, Man-in-Middle Spoofing, Hijacking | MEDIUM | LOW / NONE |



EX1015, Page 325

Oracle Corporation - Microsoft Internet Explorer provided by Compaq

File Edit View Favorites Tools Help

Back Search Favorites Media

Address http://localhost/jparser/

Search Sign In My Yahoo! Yahoo! Mail Games Personals Yahoo! Go Links

My Oracle Products Store ORACLE Downloads Contact Us Search

ADD
QUERY
SHOW TABLES

DELETE TABLES

BACK

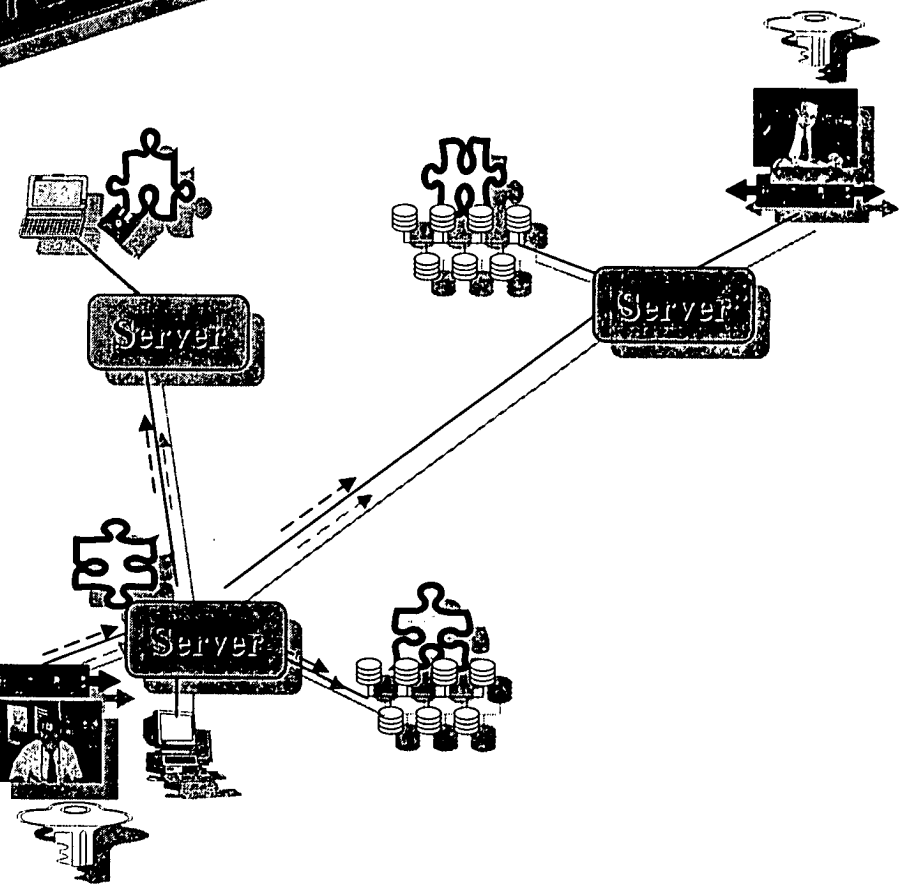
**Healthcare**

| | |
|---|---|
| Phone Number : | 2145871349 |
| Name : | James Russell |
| Address : | 10920 Elm Street |
| City : | Dallas |
| State: | TX |
| Zip code : | 75002 |
| * Social Sec # : | 781272654 |
| Age : | 51 |
| Blood Type : | A POS |
| Weight : | 215 |
| Height : | 6' 2" |
| * Medical ID : | 800972153 |
| Doctor's Name : | Vincent Paul |
| Name of Insurer : | Aetna |
| * Employer Phone Number : | 2156098214 |
| Next of Kin Name : | Carol Russell |
| Next of Kin Phone number : | 2147560912 |

SAVE CANCEL

**Data Masking**

**Parse / Encrypt / Scatter**

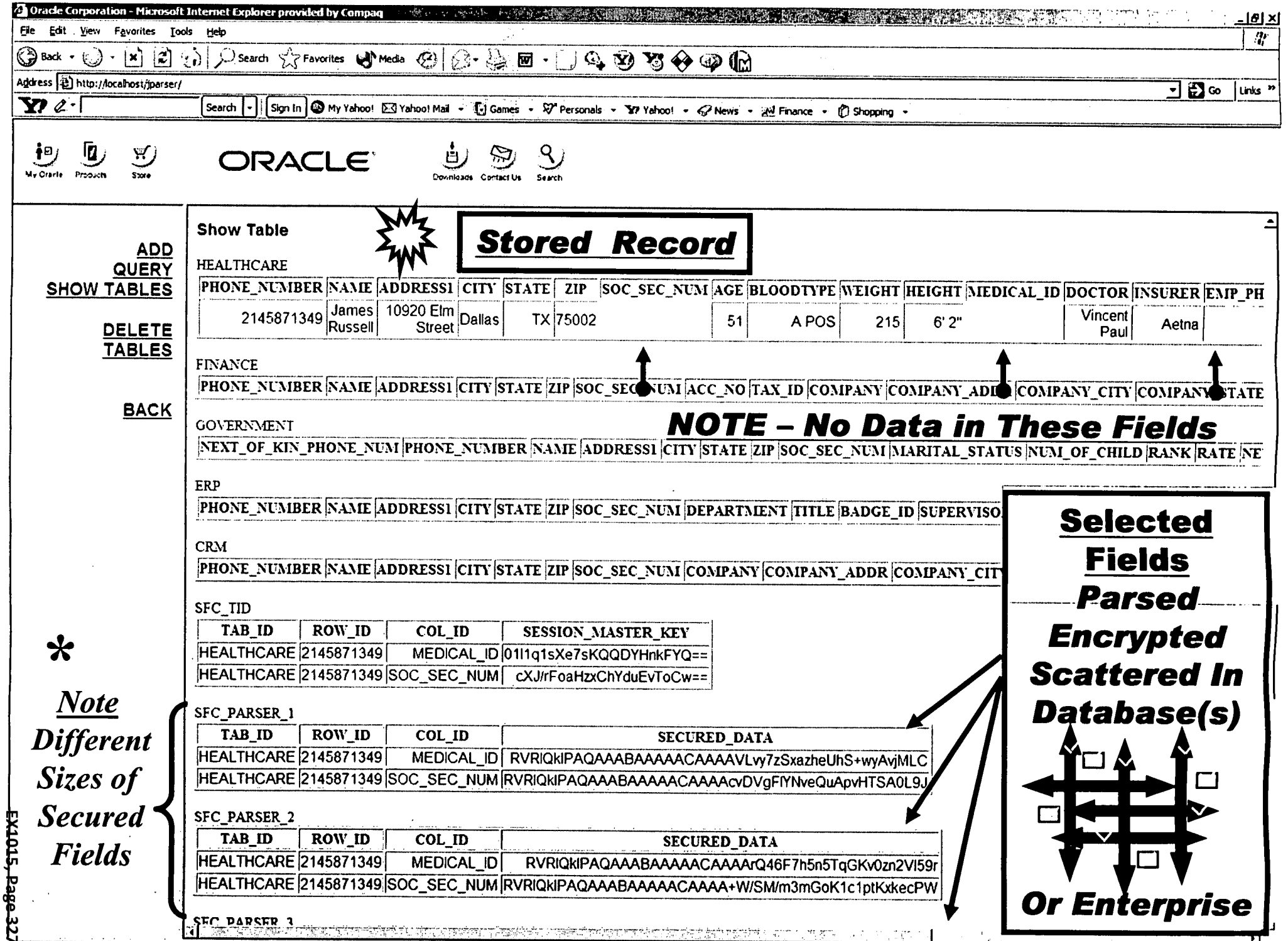



*** *Sensitive Data***

**Only selected fields are parsed then distributed**

EX1015, Page 326

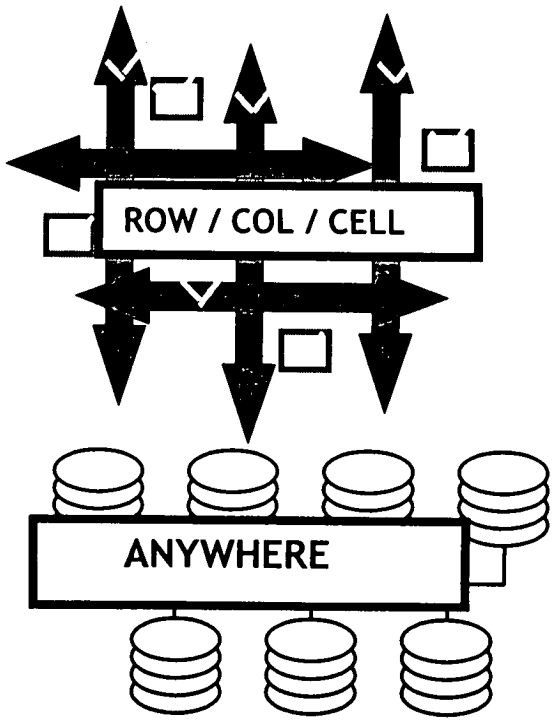
324

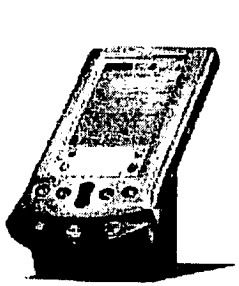

Oracle Corporation - Microsoft Internet Explorer provided by Compaq
File Edit View Favorites Tools Help
Address http://localhost/jparser/
ORACLE
My Oracle Products Store Downloads Contact Us Search
ADD
QUERY
SHOW TABLES
DELETE TABLES
BACK
Show Table
Stored Record
HEALTHCARE
PHONE_NUMBER NAME ADDRESS1 CITY STATE ZIP SOC_SEC_NUM AGE BLOODTYPE WEIGHT HEIGHT MEDICAL_ID DOCTOR INSURER EMP_PH
2145871349 James Russell 10920 Elm Street Dallas TX 75002 51 A POS 215 6' 2" Vincent Paul Aetna
FINANCE
PHONE_NUMBER NAME ADDRESS1 CITY STATE ZIP SOC_SEC_NUM ACC_NO TAX_ID COMPANY COMPANY_ADDR COMPANY_CITY COMPANY_STATE
NOTE – No Data in These Fields
GOVERNMENT
NEXT_OF_KIN_PHONE_NUM PHONE_NUMBER NAME ADDRESS1 CITY STATE ZIP SOC_SEC_NUM MARITAL_STATUS NUM_OF_CHILD RANK RATE NE
ERP
PHONE_NUMBER NAME ADDRESS1 CITY STATE ZIP SOC_SEC_NUM DEPARTMENT TITLE BADGE_ID SUPERVISO
CRM
PHONE_NUMBER NAME ADDRESS1 CITY STATE ZIP SOC_SEC_NUM COMPANY COMPANY_ADDR COMPANY_CIT
SFC_TID
TAB_ID ROW_ID COL_ID SESSION_MASTER_KEY
HEALTHCARE 2145871349 MEDICAL_ID 01l1q1sXe7sKQQDYHnkFYQ==
HEALTHCARE 2145871349 SOC_SEC_NUM cXJ/rFoaHzxChYduEvToCw==
SFC_PARSER_1
TAB_ID ROW_ID COL_ID SECURED_DATA
HEALTHCARE 2145871349 MEDICAL_ID RVRIQklPAQAAABAAAAACAAAAVLvy7zSxazheUhS+wyAvjMLC
HEALTHCARE 2145871349 SOC_SEC_NUM RVRIQklPAQAAABAAAAACAAAAcvDVgFlYNveQuApvHTSA0L9J
SFC_PARSER_2
TAB_ID ROW_ID COL_ID SECURED_DATA
HEALTHCARE 2145871349 MEDICAL_ID RVRIQklPAQAAABAAAAACAAAArQ46F7h5n5TqGKv0zn2Vl59r
HEALTHCARE 2145871349 SOC_SEC_NUM RVRIQklPAQAAABAAAAACAAAA+W/SM/m3mGoK1c1ptKxkecPW
SFC_PARSER_3
Selected Fields Parsed Encrypted Scattered In Database(s)
Or Enterprise
* Note Different Sizes of Secured Fields
Done
Local intranet

SPi
-Fault Tolerant
-Cross Platform
-Net-Centric-

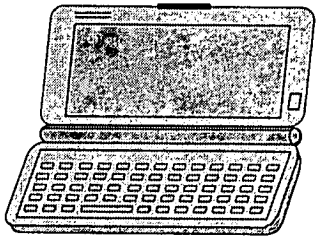
# Securing 'Data at Rest'

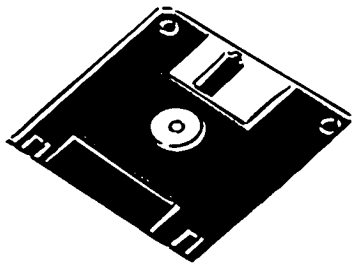
Cross Platform Interop Standard









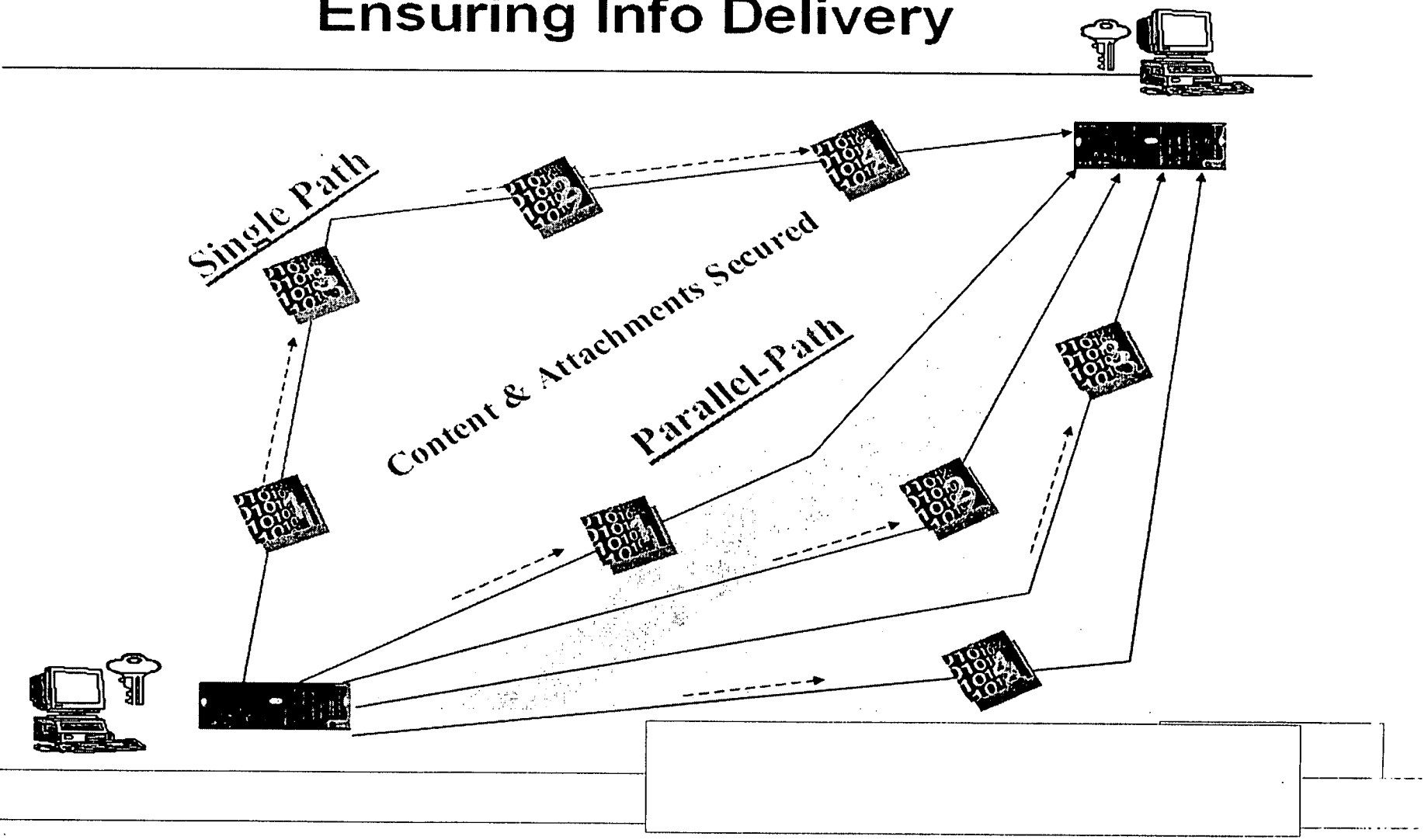


EX1015, Page 328

SPi
-Fault Tolerant
-Cross Platform
-Net-Centric

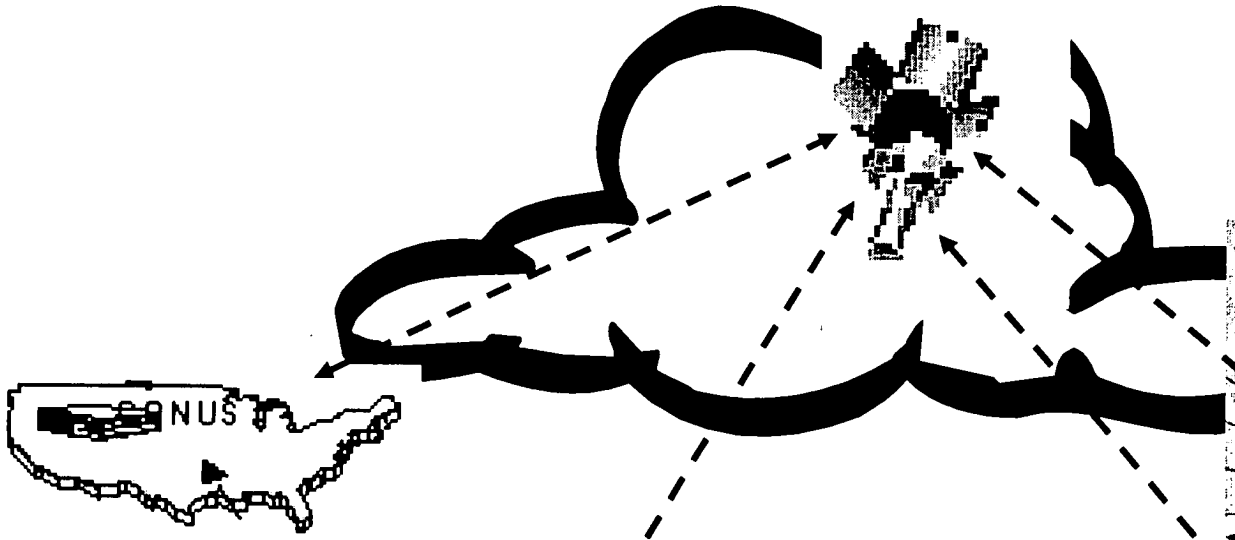
# Securing 'Data in Motion'

## Ensuring Info Delivery

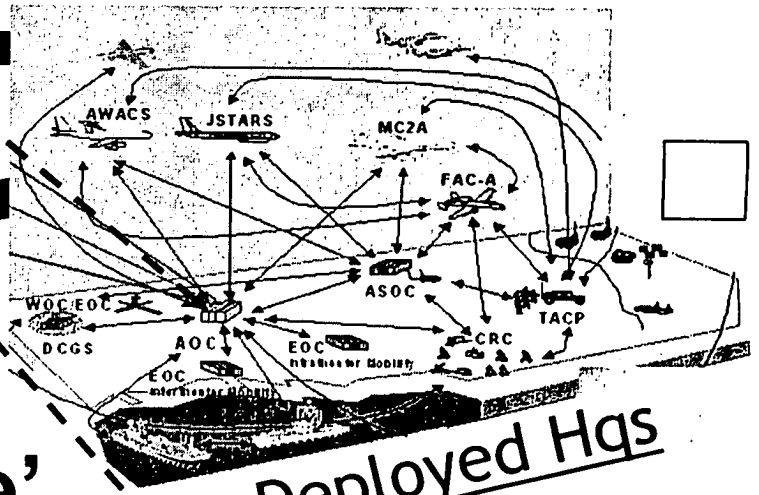





EX1015, Page 329




CONUS Hqs




Deployed Hqs

# 'Need to Share'

TS/SCI / Secret / Secret SAP / Coalition

## This Solution Is:

- Net - Centric
- Interoperable
- Secure / Parsed

Tactical Team

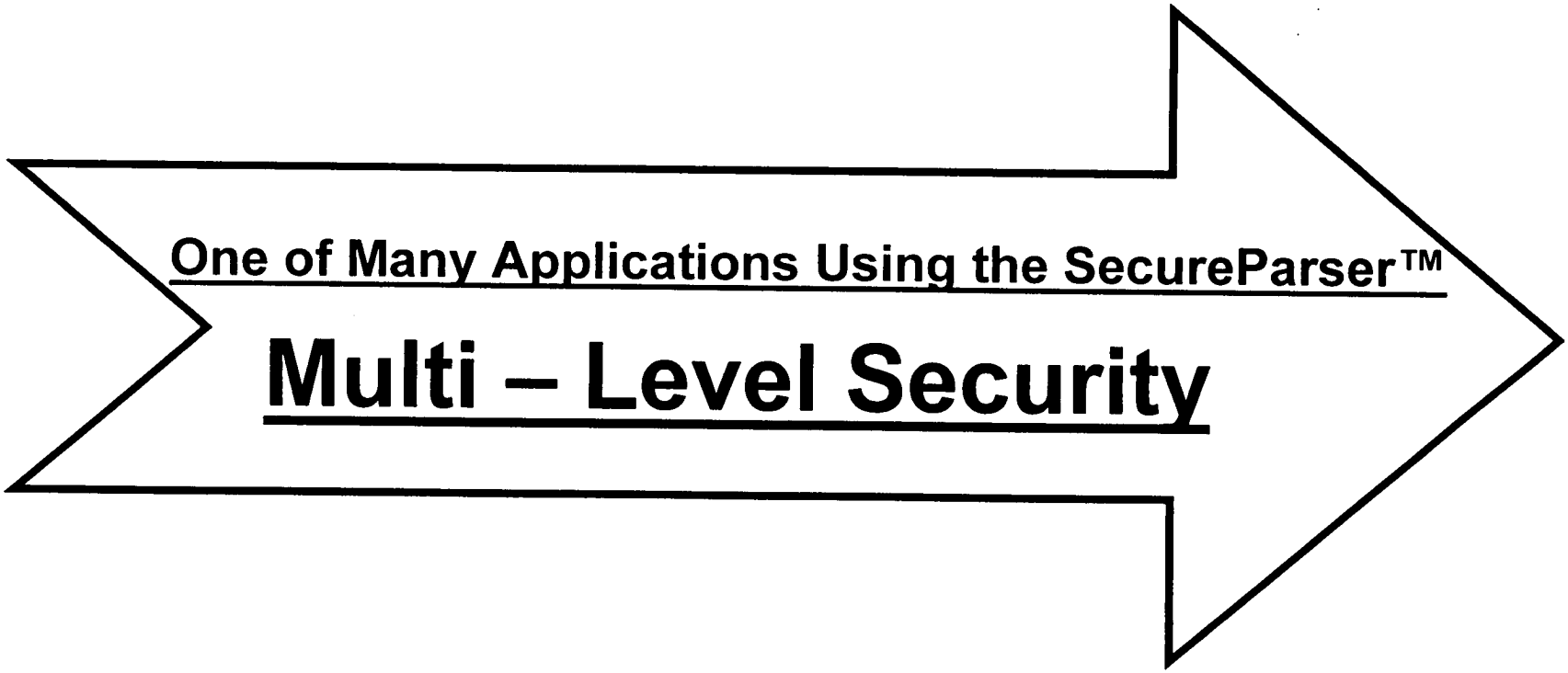


Tactical Base





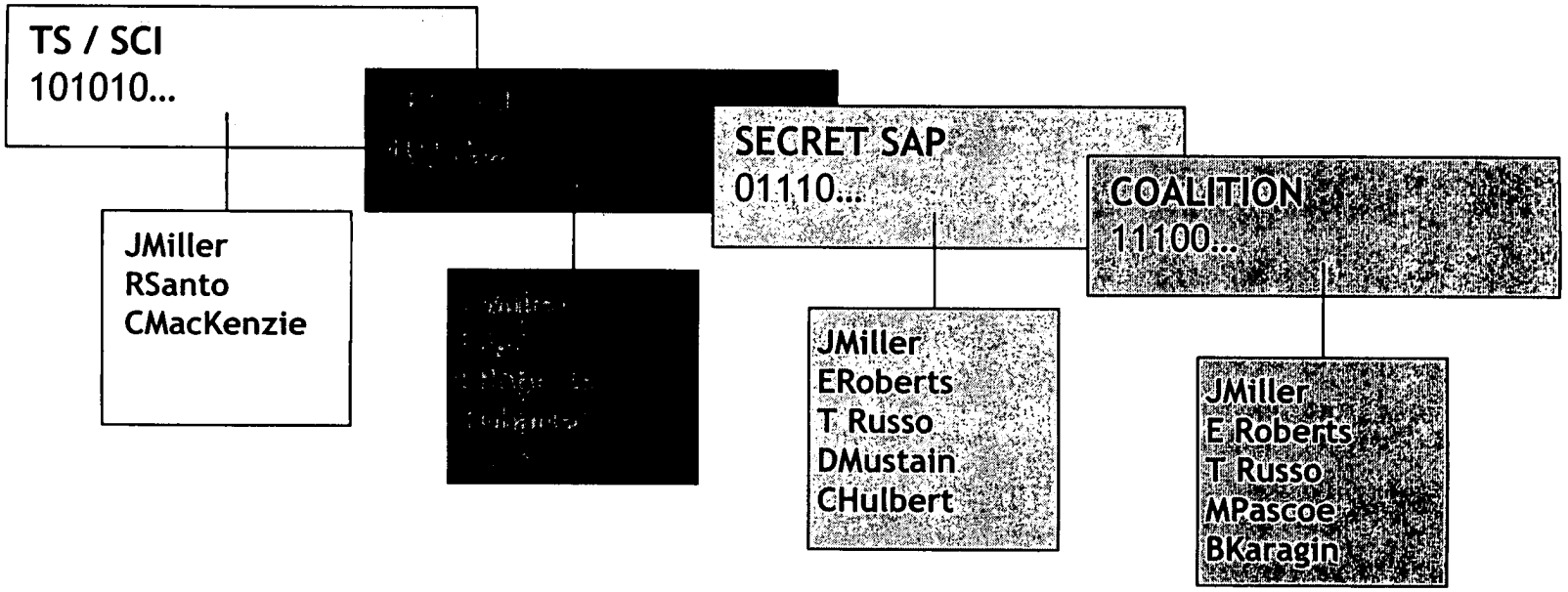
EX1015, Page 330






EX1015, Page 331

# "Need To Share"
# Multi - Level Security
# Group Key Membership

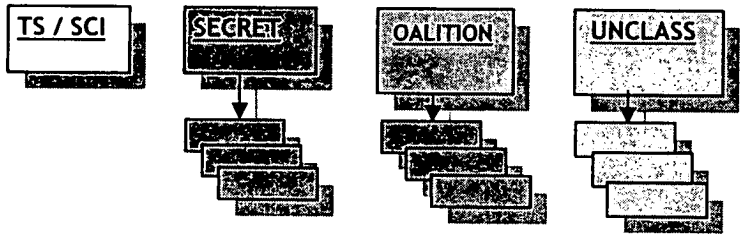



Authorized users are placed into groups allowing them use of the work group key for that classification level. Members of the Top Secret SCI group are authorized to create and read data which is secured by the SecureParser. This data cannot be accessed by any other level.

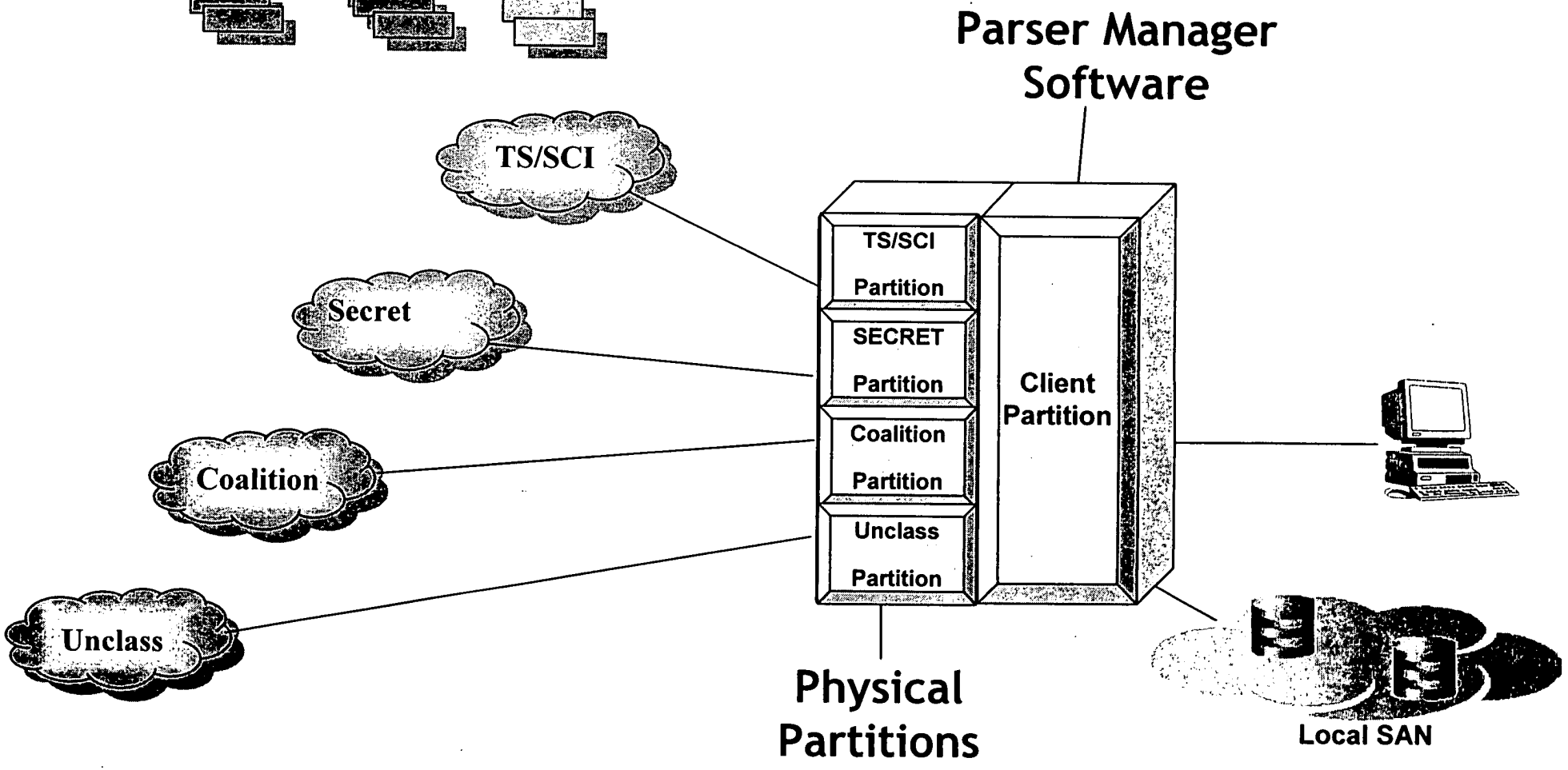


EX1015, Page 332

# Sharing Across Domains




"ParserServer", An MLS solution

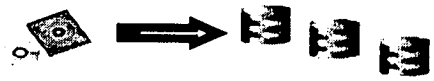




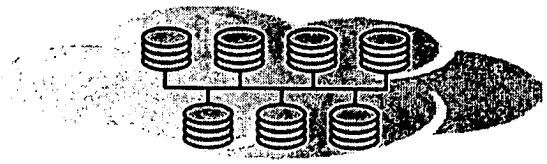

EX1015, Page 333

# Security First - Pricing Schemes "$$$$"

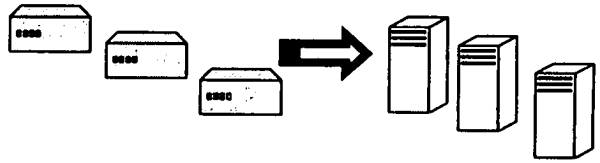
- *Data through-put: By Gigabyte*
  - *Net work storage*



- *Enterprise Based*



- *Server Based*



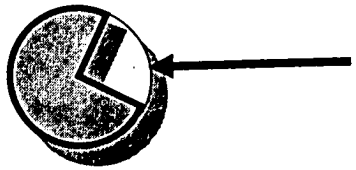
- *Client-Seat Based*



- *Site license*



- *Percentage of effort on value*




**Goal is to obtain recurring revenue whenever possible**



EX1015, Page 334

331

# **Security First** - How much data is out there

| name | number of bytes | amount of text |
|---|---|---|
| kilobyte (kB) | $2^{10}$ or 1,024 | 1/2 page |
| megabyte (mB) | $2^{20}$ or 1,048,576 | 500 pages or 1 thick book |
| gigabyte (gB) | $2^{30}$ or 1,073,741,824 | 500,000 pages or 1,000 thick books |
| terabyte (tB) | $2^{40}$ or 1,099,511,627,776 | 1,000,000 thick books |
| petabyte | $2^{50}$ or 1,125,899,906,842,624 | 180 Libraries of Congress |
| exabyte | $2^{60}$ or 1,152,921,504,606,846,976 | 180 thousand Libraries of Congress |
| zettabyte | $2^{70}$ or 1,180,591,620,717,411,303,424 | 180 million Libraries of Congress |
| yottabyte | $2^{80}$ or 1,208,925,819,614,629,174,706,176 | 180 billion Libraries of Congress |

**Security First has one effort working to secure some 23 Petabytes**



EX1015, Page 335

**Figure 1**

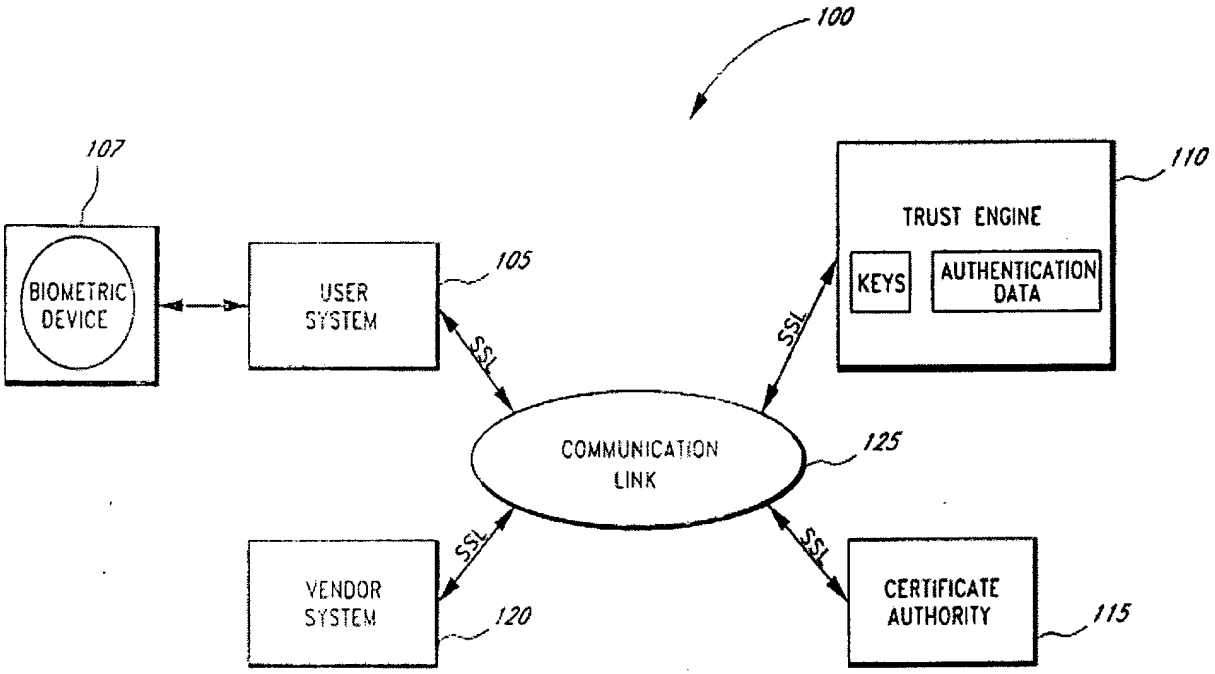

100
107
BIOMETRIC DEVICE
USER SYSTEM
105
SSL
TRUST ENGINE
110
KEYS
AUTHENTICATION DATA
SSL
COMMUNICATION LINK
125
SSL
SSL
VENDOR SYSTEM
120
CERTIFICATE AUTHORITY
115

**Figure 2**

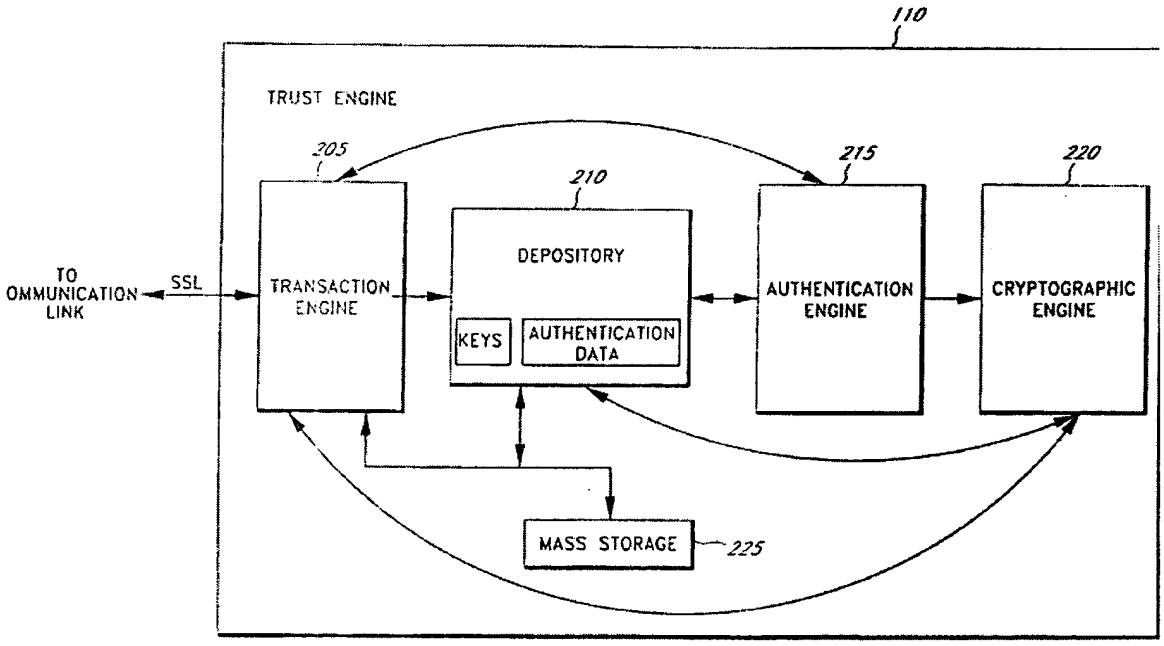

110
TRUST ENGINE
205
210
215
220
TO
OMMUNICATION
LINK
SSL
TRANSACTION
ENGINE
DEPOSITORY
KEYS
AUTHENTICATION
DATA
AUTHENTICATION
ENGINE
CRYPTOGRAPHIC
ENGINE
MASS STORAGE
225

**Figure 3**

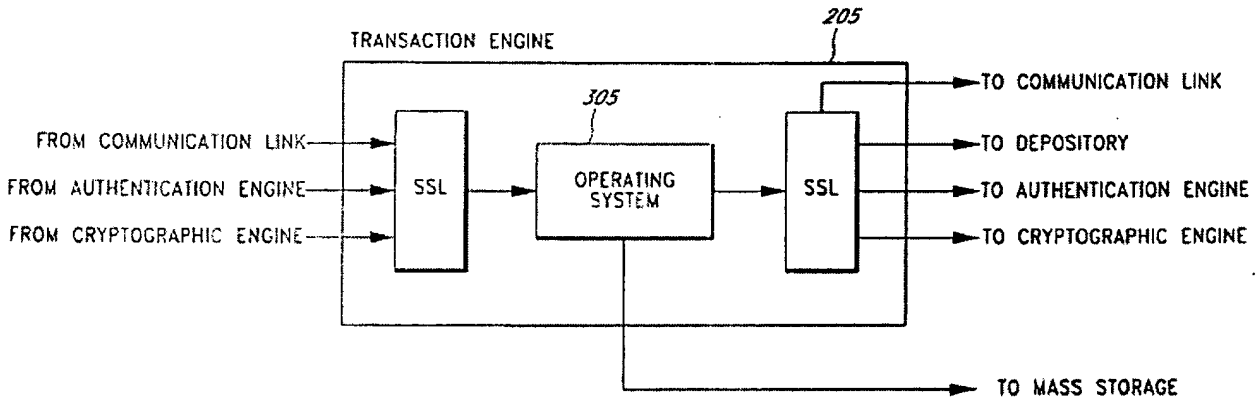

205
TRANSACTION ENGINE
305
FROM COMMUNICATION LINK
FROM AUTHENTICATION ENGINE
FROM CRYPTOGRAPHIC ENGINE
SSL
OPERATING SYSTEM
SSL
TO COMMUNICATION LINK
TO DEPOSITORY
TO AUTHENTICATION ENGINE
TO CRYPTOGRAPHIC ENGINE
TO MASS STORAGE

**Figure 4**

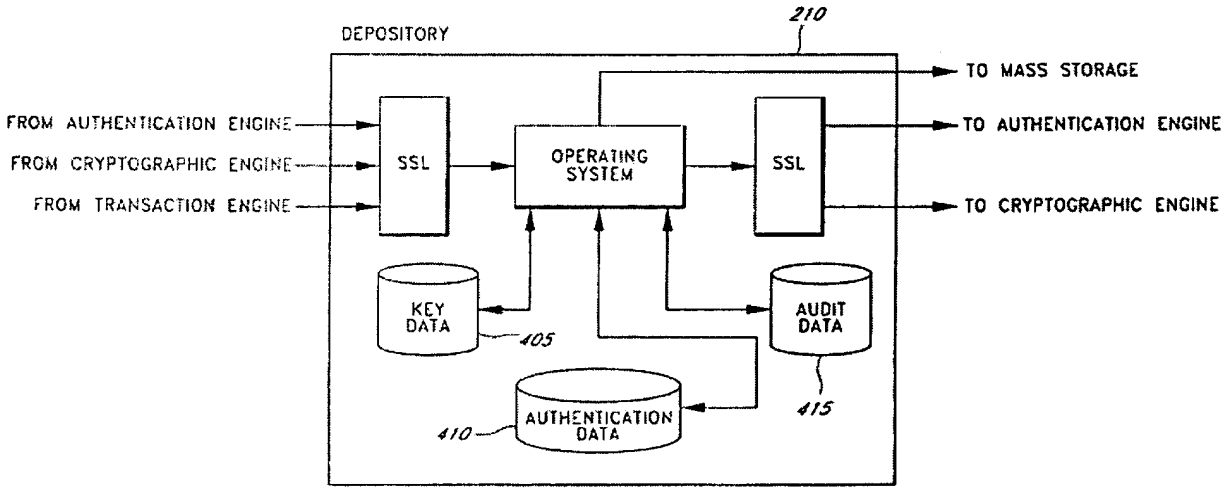

DEPOSITORY
210
FROM AUTHENTICATION ENGINE
FROM CRYPTOGRAPHIC ENGINE
FROM TRANSACTION ENGINE
SSL
OPERATING SYSTEM
SSL
TO MASS STORAGE
TO AUTHENTICATION ENGINE
TO CRYPTOGRAPHIC ENGINE
KEY DATA
405
AUDIT DATA
415
AUTHENTICATION DATA
410

**Figure 5**

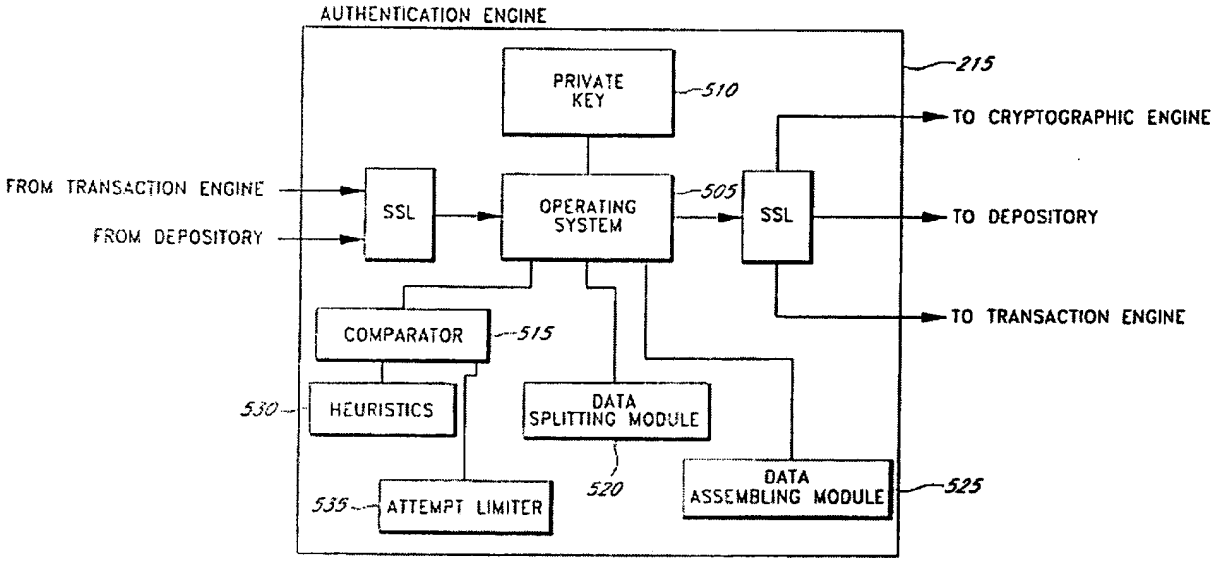

AUTHENTICATION ENGINE
215
PRIVATE KEY
510
TO CRYPTOGRAPHIC ENGINE
FROM TRANSACTION ENGINE
FROM DEPOSITORY
SSL
OPERATING SYSTEM
505
SSL
TO DEPOSITORY
TO TRANSACTION ENGINE
COMPARATOR
515
530
HEURISTICS
DATA SPLITTING MODULE
520
DATA ASSEMBLING MODULE
525
535
ATTEMPT LIMITER

**Figure 6**

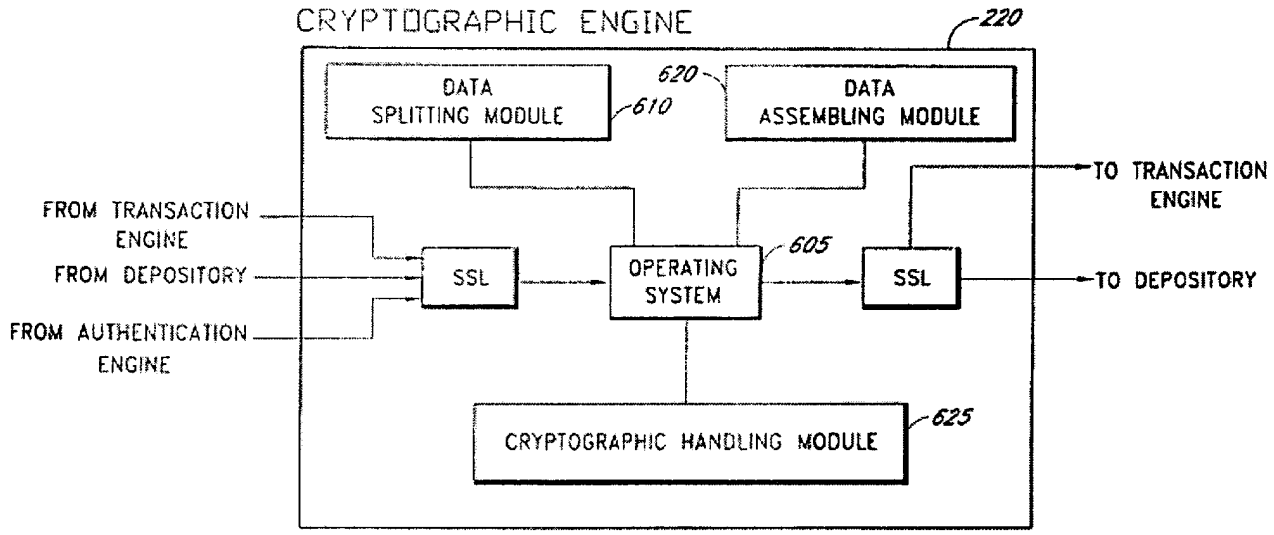

CRYPTOGRAPHIC ENGINE
220
DATA SPLITTING MODULE
610
620
DATA ASSEMBLING MODULE
FROM TRANSACTION ENGINE
FROM DEPOSITORY
FROM AUTHENTICATION ENGINE
SSL
OPERATING SYSTEM
605
SSL
TO TRANSACTION ENGINE
TO DEPOSITORY
CRYPTOGRAPHIC HANDLING MODULE
625

**Figure 7**

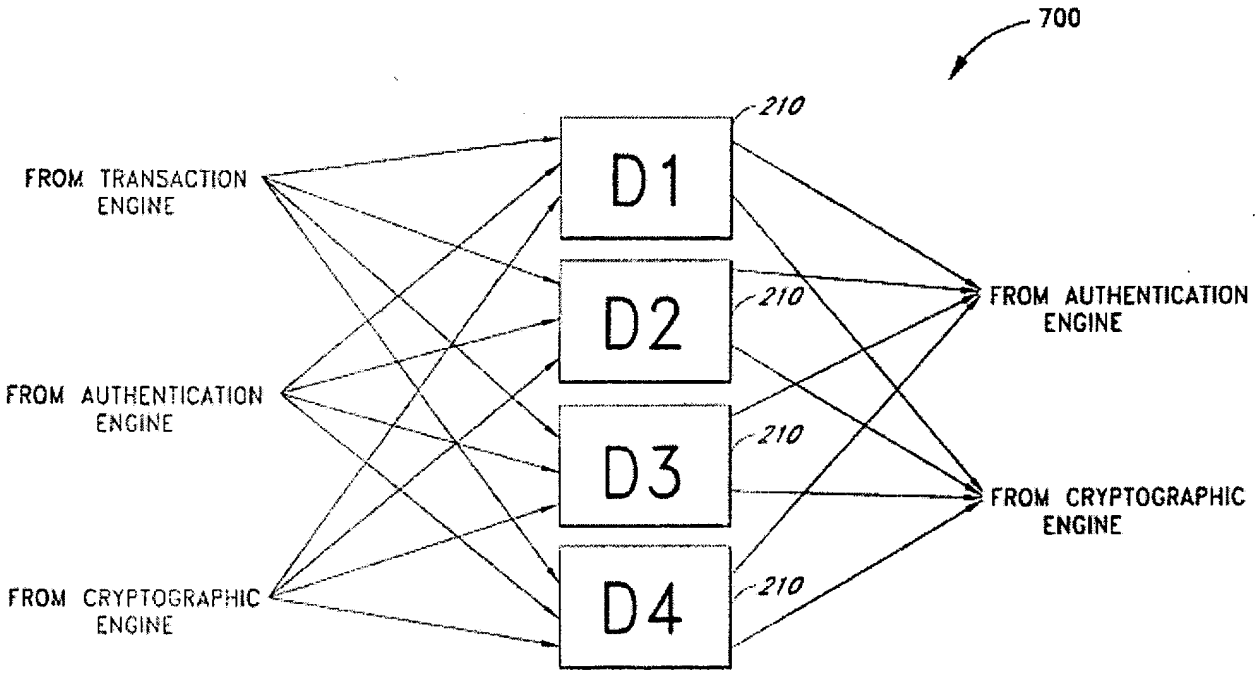

700
210
D1
D2
D3
D4
FROM TRANSACTION ENGINE
FROM AUTHENTICATION ENGINE
FROM CRYPTOGRAPHIC ENGINE
FROM AUTHENTICATION ENGINE
FROM CRYPTOGRAPHIC ENGINE

**Figure 8**

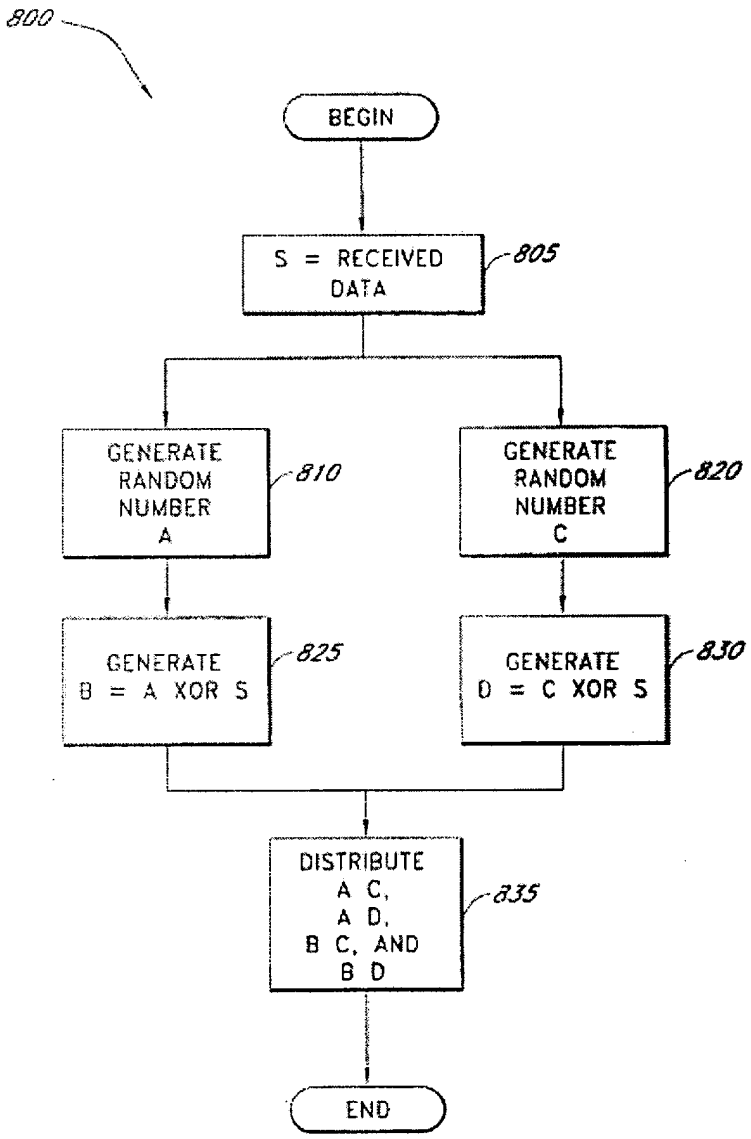
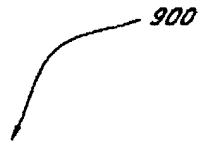

800
BEGIN
S = RECEIVED DATA
805
GENERATE RANDOM NUMBER A
810
GENERATE RANDOM NUMBER C
820
GENERATE B = A XOR S
825
GENERATE D = C XOR S
830
DISTRIBUTE A C, A D, B C, AND B D
835
END

**Figure 9, Panel A**




| ENROLLMENT DATA FLOW | | | |
|---|---|---|---|
| SEND | RECEIVE | SSL | ACTION |
| 905 USER | TRANSACTION ENGINE(TE) | 1/2 | TRANSMIT ENROLLMENT AUTHENTICATION DATA (B) AND THE USER ID (UID) ENCRYPTED WITH THE PUBLIC KEY OF THE AUTHENTICATION ENGINE (AE) AS (PUB_AE(UID,B)) |
| 915 TE | AE | FULL | FORWARD TRANSMISSION |
| 920 | | | AE DECRYPTS AND SPLITS FORWARDED DATA |
| 925 AE | THE Xih DEPOSITORY (DX) | FULL | STORE RESPECTIVE PORTION OF DATA |
| WHEN DIGITAL CERTIFICATE REQUESTED | | | |
| 930 AE | CRYPTOGRAPHIC ENGINE (CE) | FULL | REQUEST KEY GENERATION |
| 935 | | | CE GENERATES AND SPLITS KEY |
| 945 CE | TE | FULL | TRANSMIT REQUEST FOR DIGITAL CERTIFICATE |
| 950 TE | CERTIFICATION AUTHORITY (CA) | 1/2 | TRANSMIT REQUEST |
| 955 CA | TE | 1/2 | TRANSMIT DIGITAL CERTIFICATE |
| 960 TE | USER | 1/2 | TRANSMIT DIGITAL CERTIFICATE |
| TE | MS | FULL | STORE DIGITAL CERTIFICATE |
| 965 CE | DX | FULL | STORE RESPECTIVE PORTION OF KEY |

**Figure 9, Panel B**

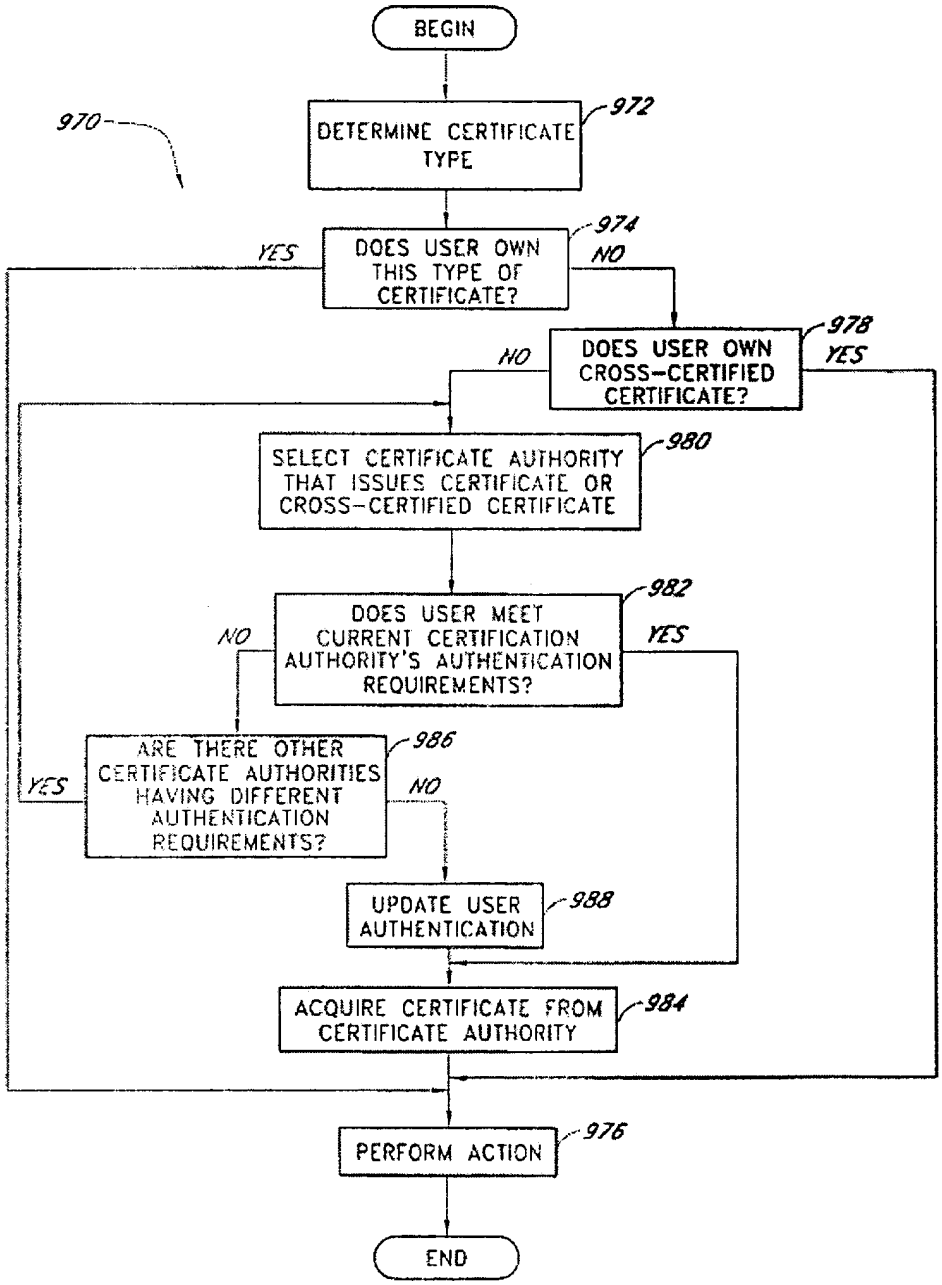


**Figure 10**




AUTHENTICATION DATA FLOW

| | SEND | RECEIVE | SSL | ACTION |
|---|---|---|---|---|
| 1005 | USER | VENDOR | 1/2 | TRANSACTION OCCURS, SUCH AS SELECTING PURCHASE |
| 1010 | VENDOR | USER | 1/2 | TRANSMIT TRANSACTION ID (TID) AND AUTHENTICATION REQUEST (AR) |
| | | | | AUTHENTICATION DATA (B') IS GATHERED FROM USER |
| 1015 | USER | TE | 1/2 | TRANSMIT TID AND B' WRAPPED IN THE PUBLIC KEY OF THE AUTHENTICATION ENGINE (AE), AS (PUB_AE(TID, B')) |
| 1020 | TE | AE | FULL | FORWARD TRANSMISSION |
| | | | | ENROLLMENT AUTHENTICATION DATA (B) IS REQUESTED AND GATHERED |
| 1025 | VENDOR | TRANSACTION ENGINE (TE) | FULL | TRANSMITS TID, AR |
| 1030 | TE | MASS STORAGE(MS) | FULL | CREATE RECORD IN DATABASE |
| 1035 | TE | THE Xth DEPOSITORY(DX) | FULL | UID, TID |
| 1040 | DX | AE | FULL | TRANSMIT THE TID AND THE PORTION OF THE AUTHENTICATION DATA STORED AT ENROLLMENT (BX) AS (PUB_AE(TID, BX)) |
| 1045 | | | | AE ASSEMBLES B AND COMPARES TO B' |
| 1050 | AE | TE | FULL | TID, THE FILLED IN AR |
| | TE | VENDOR | FULL | TID, YES/NO |
| 1055 | TE | USER | 1/2 | TID, CONFIRMATION MESSAGE |

**Figure 11**

1100

| SIGNING DATA FLOW | | | |
|---|---|---|---|
| SEND | RECEIVE | SSL | ACTION |
| USER | VENDOR | 1/2 | TRANSACTION OCCURS, SUCH AS AGREEING ON A DEAL |
| VENDOR | USER | 1/2 | TRANSMIT TRANSACTION IDENTIFICATION NUMBER (TID), AUTHENTICATION REQUEST (AR), AND AGREEMENT OR MESSAGE (M) |
| | | | CURRENT AUTHENTICATION DATA (B') AND A HASH OF THE MESSAGE RECEIVED BY THE USER (h(M')) IS GATHERED FROM USER |
| USER | TE | 1/2 | TRANSMIT TID, B', AR, AND h(M') WRAPPED IN THE PUBLIC KEY OF THE AUTHENTICATION ENGINE (AE) AS (PUB_AE(TID, B', h(M')) |
| TE | AE | FULL | FORWARD TRANSMISSION |
| | | | GATHER ENROLLMENT AUTHENTICATION DATA |
| VENDOR | TRANSACTION ENGINE (TE) | FULL | TRANSMITS UID, TID, AR, AND A HASH OF THE MESSAGE (h(M)). |
| TE | MASS STORAGE (MS) | FULL | CREATE RECORD IN DATABASE |
| TE | THE Xth DEPOSITORY(DX) | FULL | UID, TID |
| DX | AE | FULL | TRANSMIT THE TID AND THE PORTION OF THE AUTHENTICATION DATA STORED AT ENROLLMENT (BX),AS (PUB_AE(TID, BX)) |
| | | | THE ORIGINAL VENDOR MESSAGE IS TRANSMITTED TO THE AE |
| TE | AE | FULL | TRANSMIT h(M) |
| | | | AE ASSEMBLES B, COMPARES TO B' AND COMPARES h(M) TO h(M') |
| AE | CRYPTOGRAPHIC ENGINE (CE) | FULL | REQUEST FOR DIGITAL SIGNATURE AND A MESSAGE TO BE SIGNED, FOR EXAMPLE, THE HASHED MESSAGE |
| AE | DX | FULL | TID, SIGNING UID |
| DX | CE | FULL | TRANSMIT THE PORTION OF THE CRYPTOGRAPHIC KEY CORRESPONDING TO THE SIGNING PARTY |
| | | | CE ASSEMBLES KEY AND SIGNS |
| CE | AE | FULL | TRANSMIT THE DIGITAL SIGNATURE (S) OF SIGNING PARTY |
| AE | TE | FULL | TID, THE FILLED IN AR, h(M), AND S |
| TE | VENDOR | FULL | TID, A RECEIPT=(TID, YES/NO, AND S), AND THE DIGITAL SIGNATURE OF THE TRUST ENGINE, FOR EXAMPLE, A HASH OF THE RECEIPT ENCRYPTED WITH THE TRUST ENGINE'S PRIVATE KEY (Priv_TE(h(RECEIPT))) |
| TE | USER | 1/2 | TID, CONFIRMATION MESSAGE |

1103 1105 1110 1115 1120 1125 1130 1135 1140

**Figure 12**

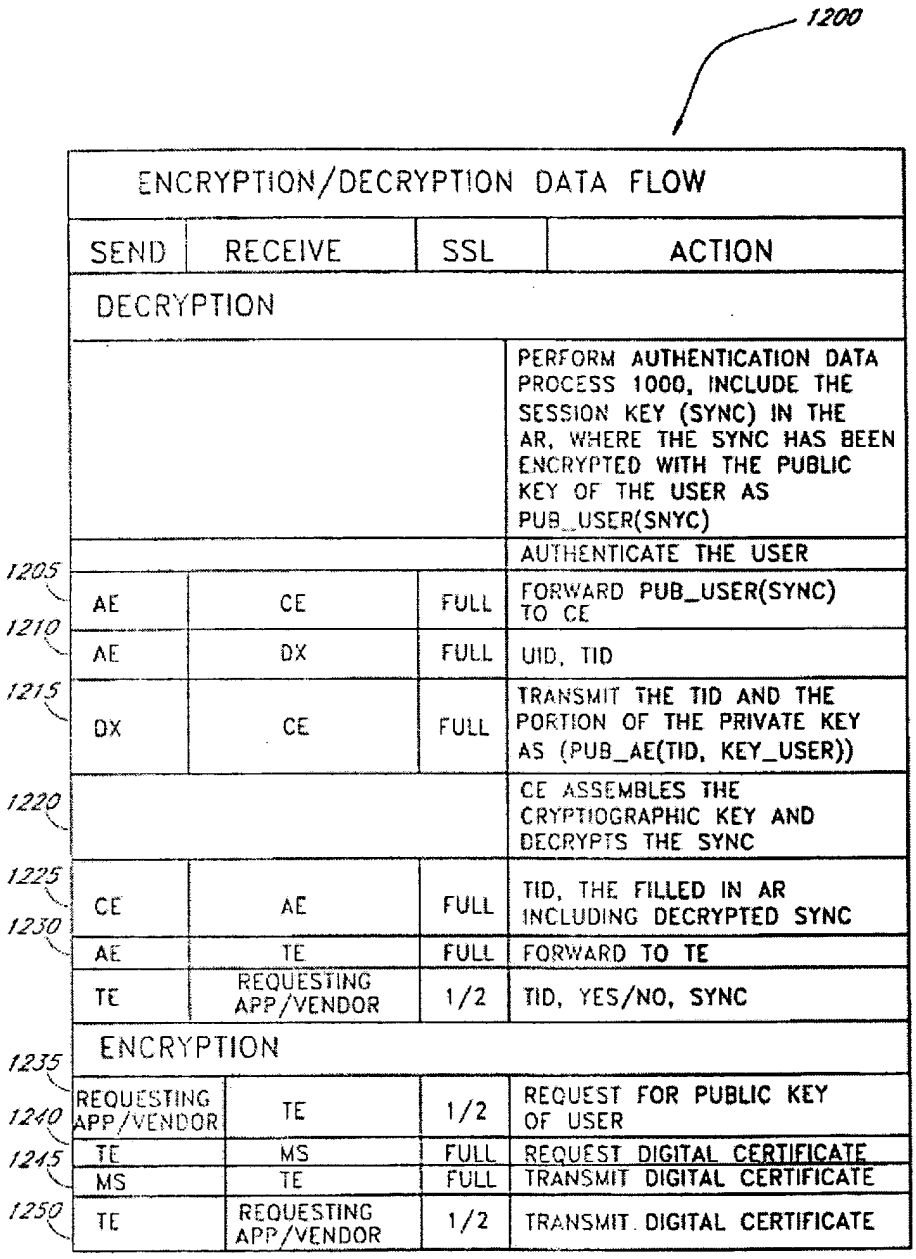

1200
ENCRYPTION/DECRYPTION DATA FLOW
SEND
RECEIVE
SSL
ACTION
DECRYPTION
PERFORM AUTHENTICATION DATA PROCESS 1000, INCLUDE THE SESSION KEY (SYNC) IN THE AR, WHERE THE SYNC HAS BEEN ENCRYPTED WITH THE PUBLIC KEY OF THE USER AS PUB_USER(SNYC)
AUTHENTICATE THE USER
1205
AE
CE
FULL
FORWARD PUB_USER(SYNC) TO CE
1210
AE
DX
FULL
UID, TID
1215
DX
CE
FULL
TRANSMIT THE TID AND THE PORTION OF THE PRIVATE KEY AS (PUB_AE(TID, KEY_USER))
1220
CE ASSEMBLES THE CRYPTIOGRAPHIC KEY AND DECRYPTS THE SYNC
1225
CE
AE
FULL
TID, THE FILLED IN AR INCLUDING DECRYPTED SYNC
1230
AE
TE
FULL
FORWARD TO TE
TE
REQUESTING APP/VENDOR
1/2
TID, YES/NO, SYNC
ENCRYPTION
1235
REQUESTING APP/VENDOR
TE
1/2
REQUEST FOR PUBLIC KEY OF USER
1240
TE
MS
FULL
REQUEST DIGITAL CERTIFICATE
1245
MS
TE
FULL
TRANSMIT DIGITAL CERTIFICATE
1250
TE
REQUESTING APP/VENDOR
1/2
TRANSMIT DIGITAL CERTIFICATE

**Figure 13**

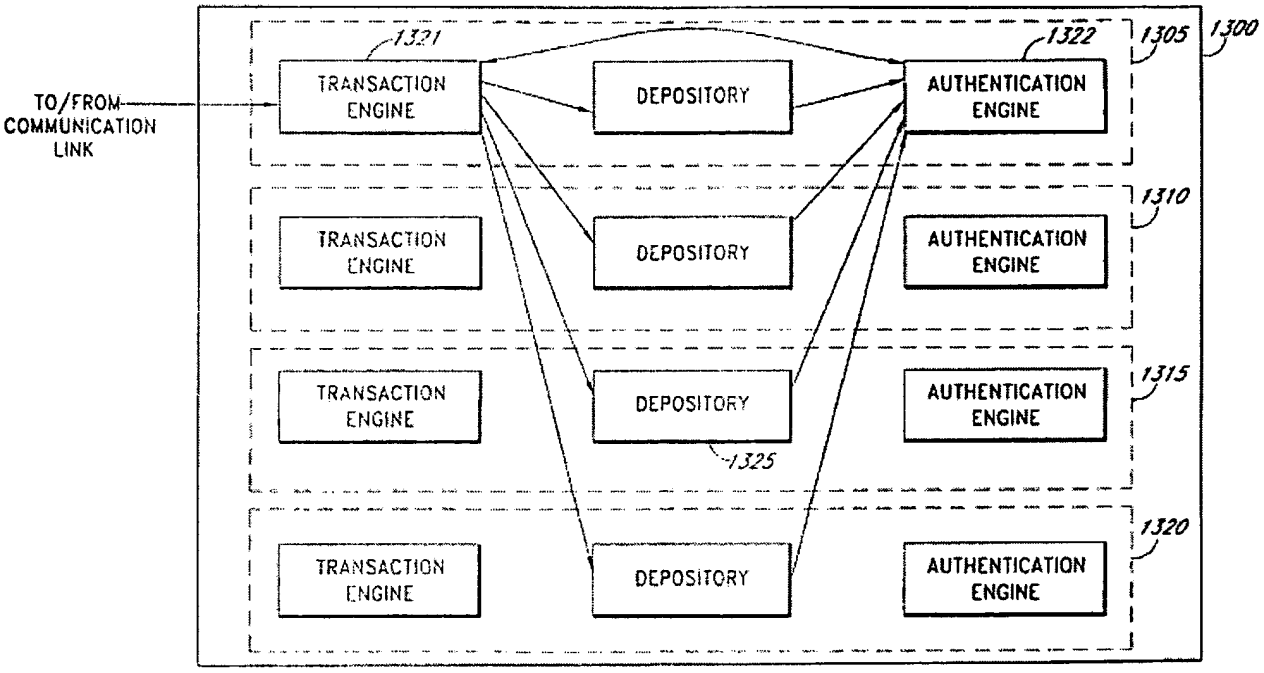

TO/FROM COMMUNICATION LINK
1321
TRANSACTION ENGINE
DEPOSITORY
1322
AUTHENTICATION ENGINE
1305
1300
TRANSACTION ENGINE
DEPOSITORY
AUTHENTICATION ENGINE
1310
TRANSACTION ENGINE
DEPOSITORY
1325
AUTHENTICATION ENGINE
1315
TRANSACTION ENGINE
DEPOSITORY
AUTHENTICATION ENGINE
1320

**Figure 14**

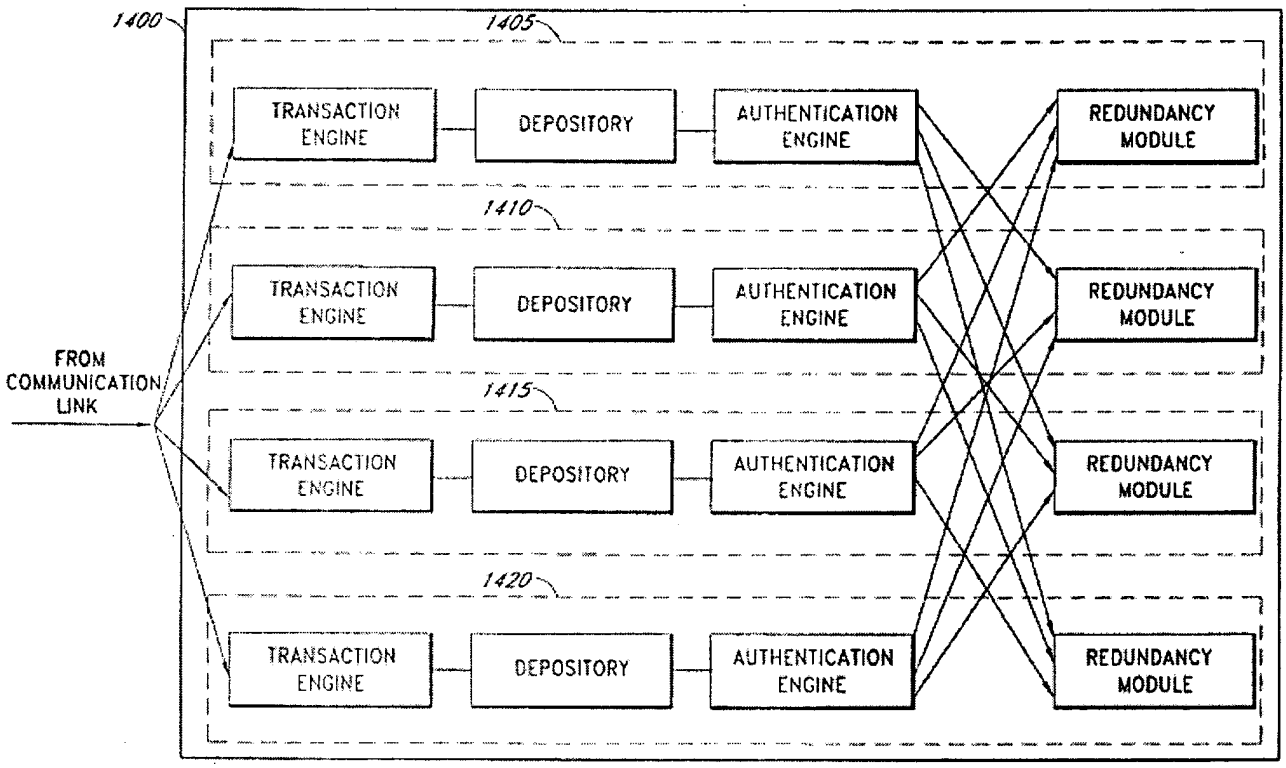

1400
1405
TRANSACTION ENGINE
DEPOSITORY
AUTHENTICATION ENGINE
REDUNDANCY MODULE
1410
TRANSACTION ENGINE
DEPOSITORY
AUTHENTICATION ENGINE
REDUNDANCY MODULE
FROM COMMUNICATION LINK
1415
TRANSACTION ENGINE
DEPOSITORY
AUTHENTICATION ENGINE
REDUNDANCY MODULE
1420
TRANSACTION ENGINE
DEPOSITORY
AUTHENTICATION ENGINE
REDUNDANCY MODULE

**Figure 15**

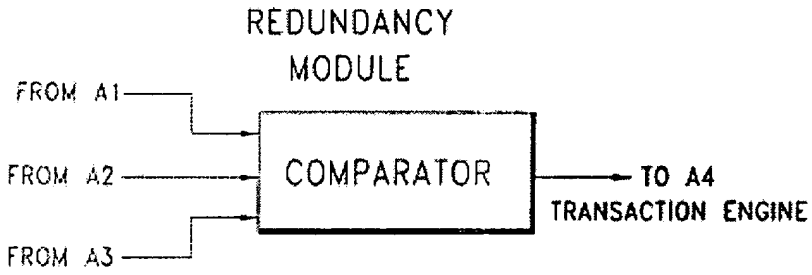

REDUNDANCY
MODULE
FROM A1
FROM A2
FROM A3
COMPARATOR
TO A4
TRANSACTION ENGINE

**Figure 16**

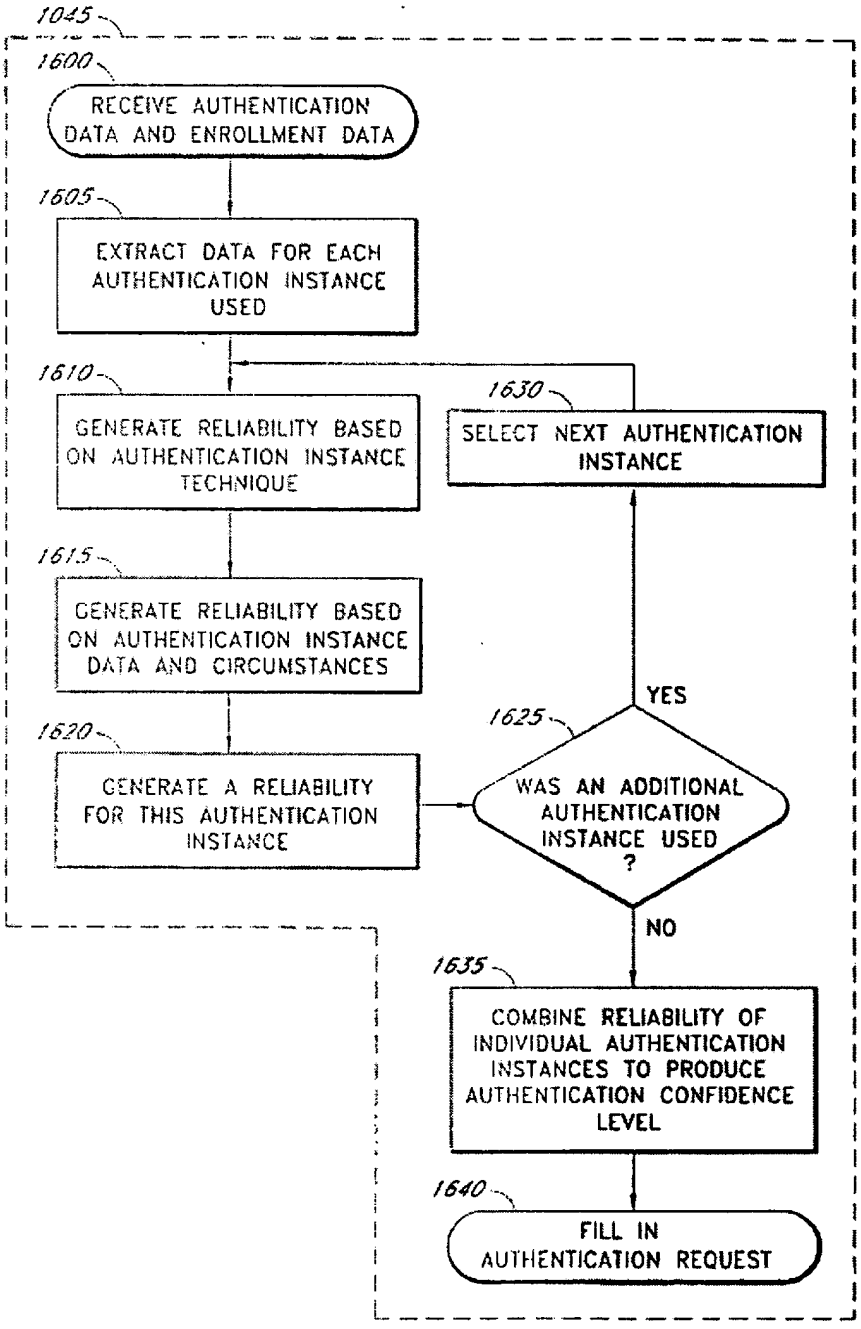

1045
1600
RECEIVE AUTHENTICATION DATA AND ENROLLMENT DATA
1605
EXTRACT DATA FOR EACH AUTHENTICATION INSTANCE USED
1610
GENERATE RELIABILITY BASED ON AUTHENTICATION INSTANCE TECHNIQUE
1615
GENERATE RELIABILITY BASED ON AUTHENTICATION INSTANCE DATA AND CIRCUMSTANCES
1620
GENERATE A RELIABILITY FOR THIS AUTHENTICATION INSTANCE
1625
WAS AN ADDITIONAL AUTHENTICATION INSTANCE USED ?
YES
NO
1630
SELECT NEXT AUTHENTICATION INSTANCE
1635
COMBINE RELIABILITY OF INDIVIDUAL AUTHENTICATION INSTANCES TO PRODUCE AUTHENTICATION CONFIDENCE LEVEL
1640
FILL IN AUTHENTICATION REQUEST

**Figure 17**

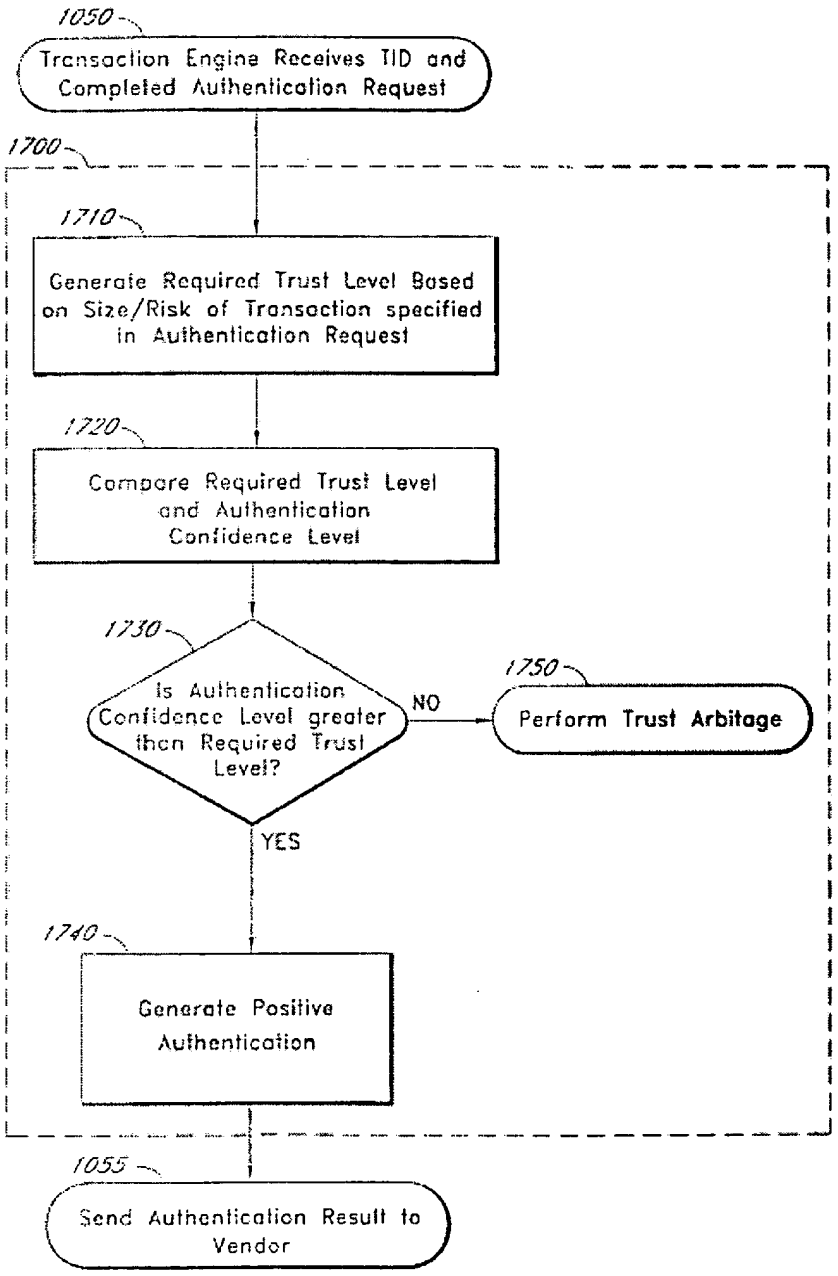

1050
Transaction Engine Receives TID and Completed Authentication Request
1700
1710
Generate Required Trust Level Based on Size/Risk of Transaction specified in Authentication Request
1720
Compare Required Trust Level and Authentication Confidence Level
1730
Is Authentication Confidence Level greater than Required Trust Level?
NO
1750
Perform Trust Arbitage
YES
1740
Generate Positive Authentication
1055
Send Authentication Result to Vendor

**Figure 18**

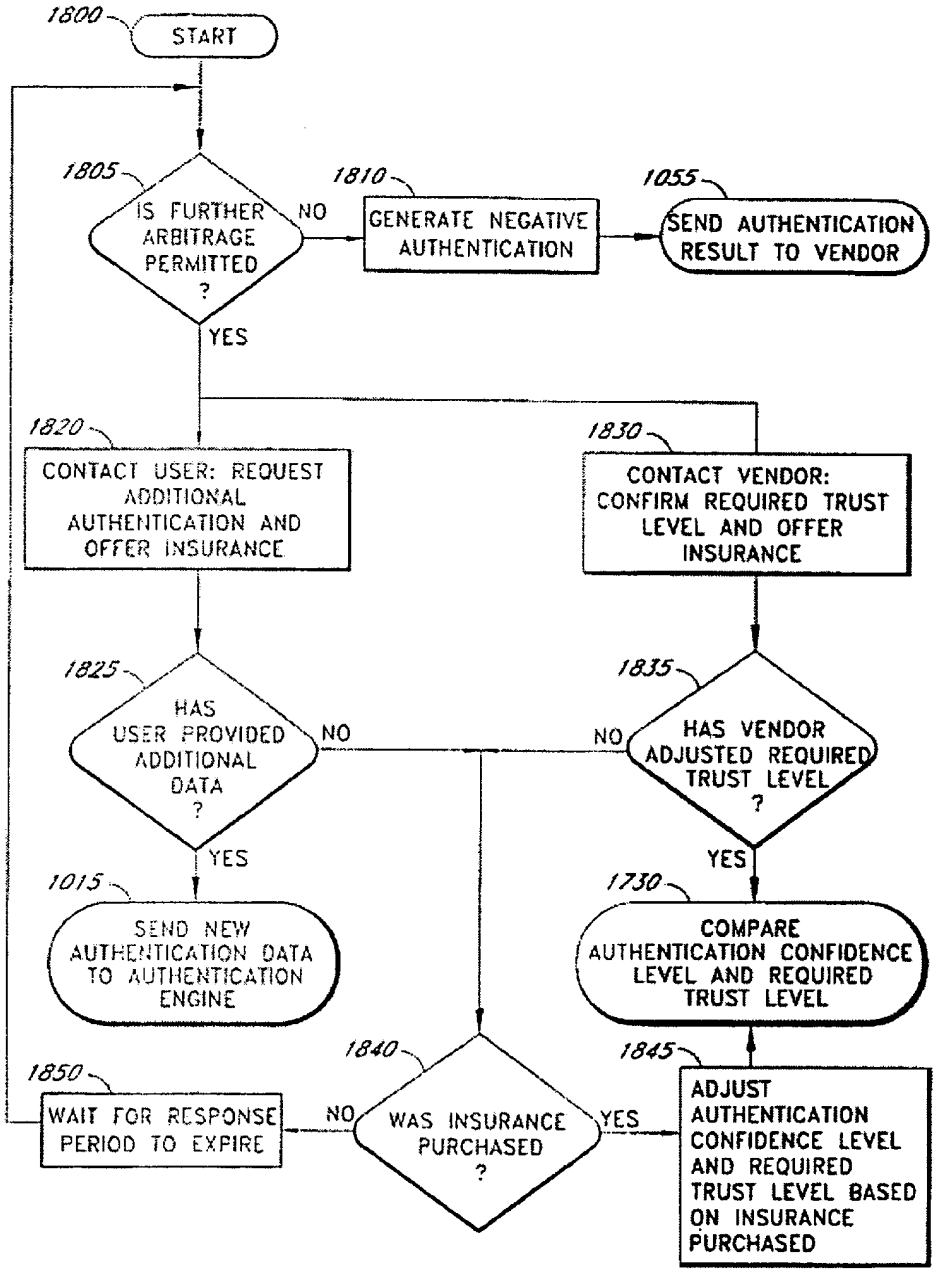

1800
START
1805
IS FURTHER ARBITRAGE PERMITTED ?
NO
1810
GENERATE NEGATIVE AUTHENTICATION
1055
SEND AUTHENTICATION RESULT TO VENDOR
YES
1820
CONTACT USER: REQUEST ADDITIONAL AUTHENTICATION AND OFFER INSURANCE
1830
CONTACT VENDOR: CONFIRM REQUIRED TRUST LEVEL AND OFFER INSURANCE
1825
HAS USER PROVIDED ADDITIONAL DATA ?
NO
1835
HAS VENDOR ADJUSTED REQUIRED TRUST LEVEL ?
NO
YES
YES
1015
SEND NEW AUTHENTICATION DATA TO AUTHENTICATION ENGINE
1730
COMPARE AUTHENTICATION CONFIDENCE LEVEL AND REQUIRED TRUST LEVEL
1840
1845
1850
WAIT FOR RESPONSE PERIOD TO EXPIRE
NO
WAS INSURANCE PURCHASED ?
YES
ADJUST AUTHENTICATION CONFIDENCE LEVEL AND REQUIRED TRUST LEVEL BASED ON INSURANCE PURCHASED

**Figure 19**

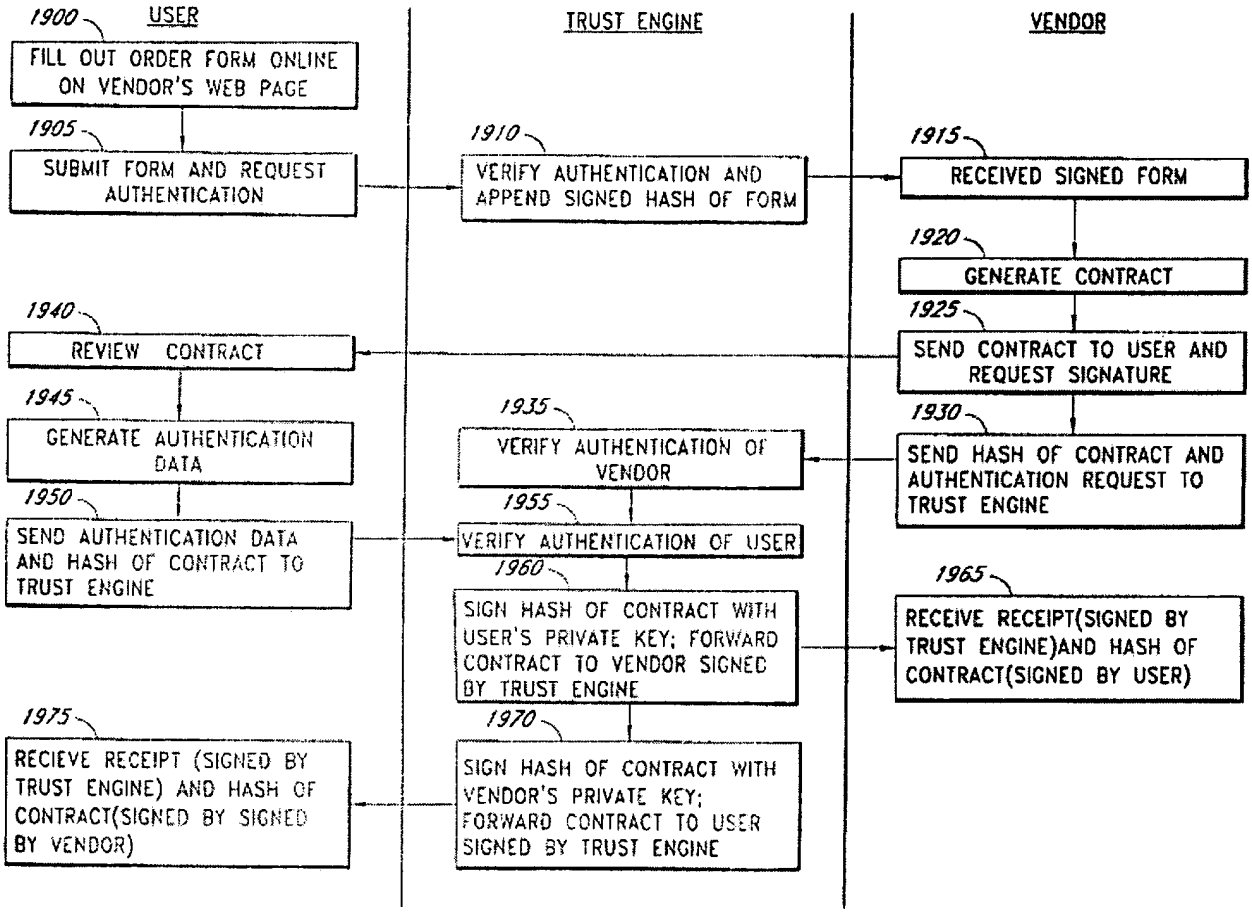

1900
USER
FILL OUT ORDER FORM ONLINE ON VENDOR'S WEB PAGE
1905
SUBMIT FORM AND REQUEST AUTHENTICATION
1940
REVIEW CONTRACT
1945
GENERATE AUTHENTICATION DATA
1950
SEND AUTHENTICATION DATA AND HASH OF CONTRACT TO TRUST ENGINE
1975
RECIEVE RECEIPT (SIGNED BY TRUST ENGINE) AND HASH OF CONTRACT(SIGNED BY SIGNED BY VENDOR)
TRUST ENGINE
1910
VERIFY AUTHENTICATION AND APPEND SIGNED HASH OF FORM
1935
VERIFY AUTHENTICATION OF VENDOR
1955
VERIFY AUTHENTICATION OF USER
1960
SIGN HASH OF CONTRACT WITH USER'S PRIVATE KEY; FORWARD CONTRACT TO VENDOR SIGNED BY TRUST ENGINE
1970
SIGN HASH OF CONTRACT WITH VENDOR'S PRIVATE KEY; FORWARD CONTRACT TO USER SIGNED BY TRUST ENGINE
VENDOR
1915
RECEIVED SIGNED FORM
1920
GENERATE CONTRACT
1925
SEND CONTRACT TO USER AND REQUEST SIGNATURE
1930
SEND HASH OF CONTRACT AND AUTHENTICATION REQUEST TO TRUST ENGINE
1965
RECEIVE RECEIPT(SIGNED BY TRUST ENGINE)AND HASH OF CONTRACT(SIGNED BY USER)

**Figure 20**

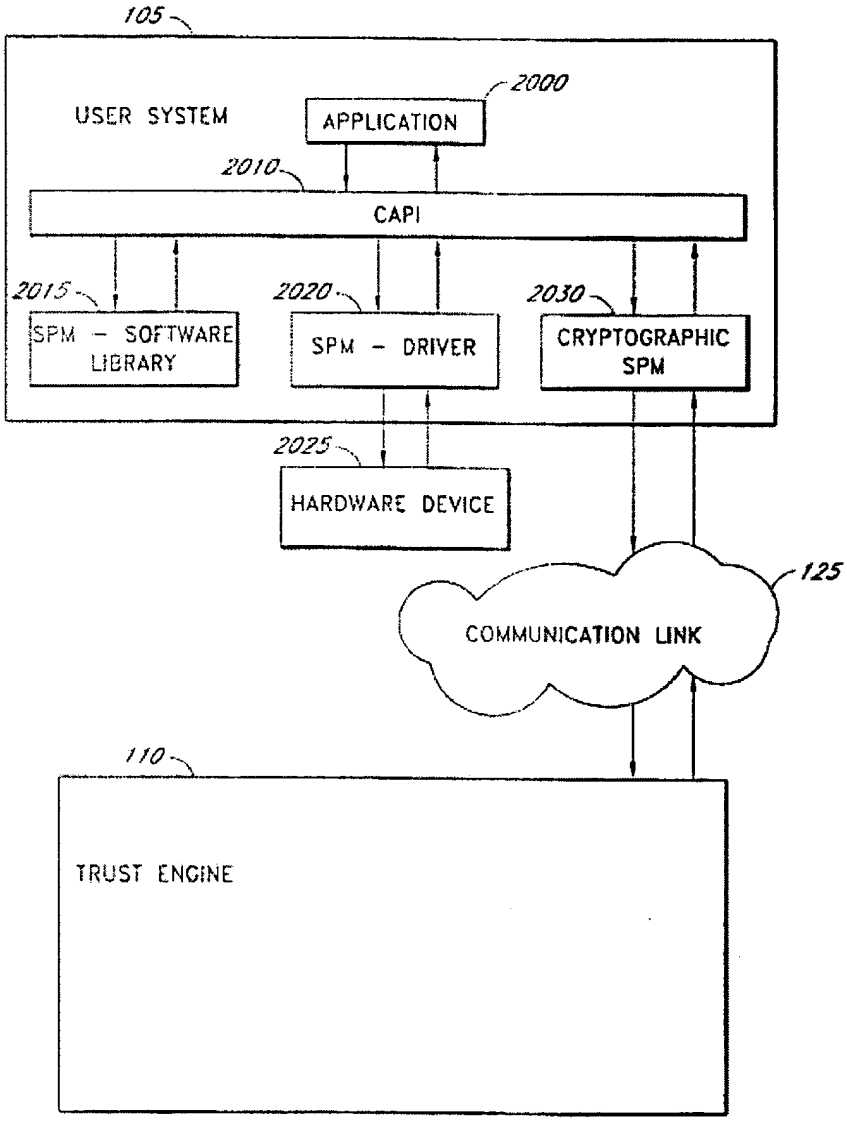

105
USER SYSTEM
2000
APPLICATION
2010
CAPI
2015
SPM – SOFTWARE LIBRARY
2020
SPM – DRIVER
2030
CRYPTOGRAPHIC SPM
2025
HARDWARE DEVICE
125
COMMUNICATION LINK
110
TRUST ENGINE

**Figure 21**

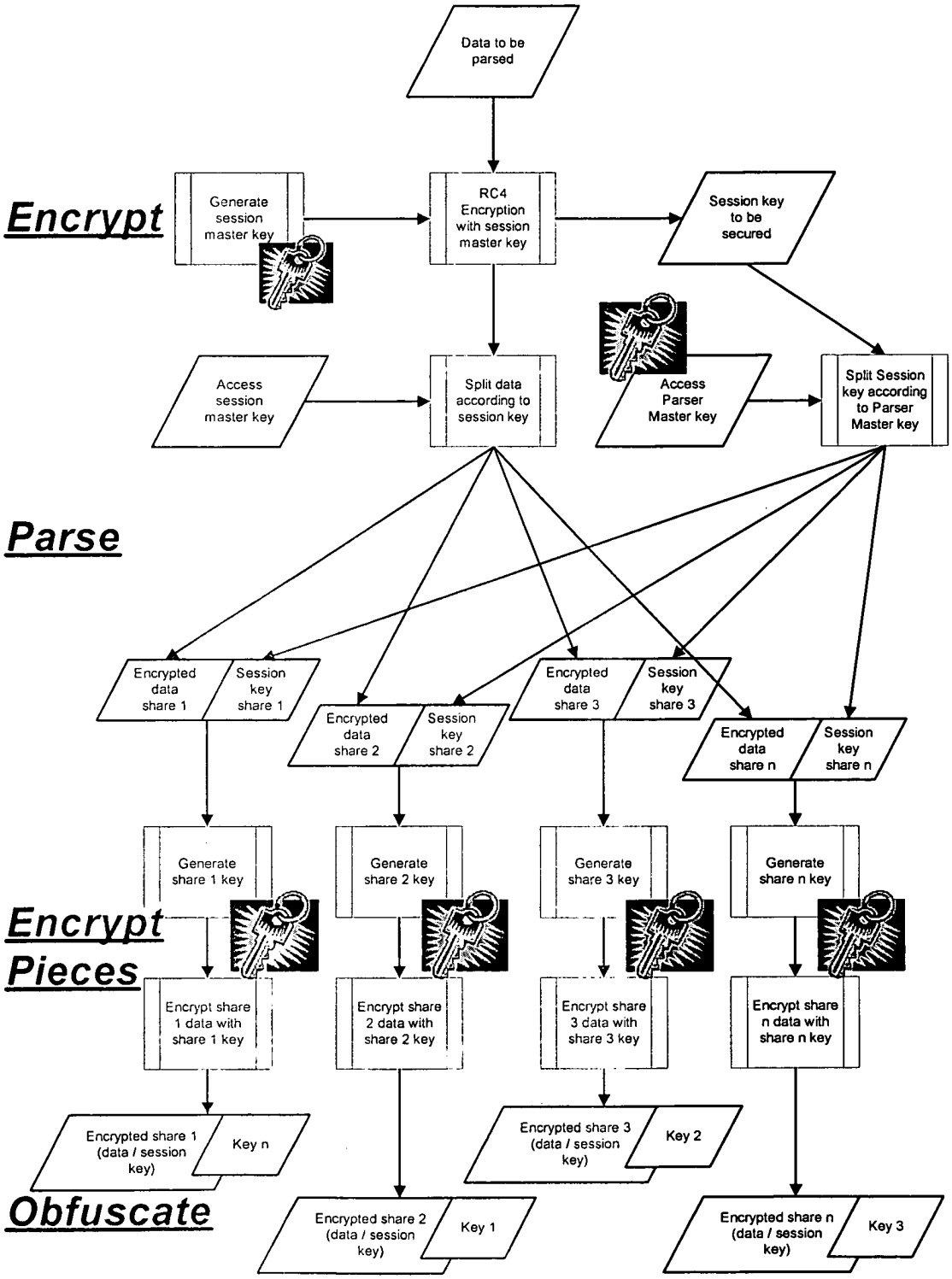

Data to be parsed
Encrypt
Generate session master key
RC4 Encryption with session master key
Session key to be secured
Access session master key
Split data according to session key
Access Parser Master key
Split Session key according to Parser Master key
Parse
Encrypted data share 1
Session key share 1
Encrypted data share 2
Session key share 2
Encrypted data share 3
Session key share 3
Encrypted data share n
Session key share n
Encrypt Pieces
Generate share 1 key
Generate share 2 key
Generate share 3 key
Generate share n key
Encrypt share 1 data with share 1 key
Encrypt share 2 data with share 2 key
Encrypt share 3 data with share 3 key
Encrypt share n data with share n key
Obfuscate
Encrypted share 1 (data / session key)
Key n
Encrypted share 2 (data / session key)
Key 1
Encrypted share 3 (data / session key)
Key 2
Encrypted share n (data / session key)
Key 3

**Figure 22**

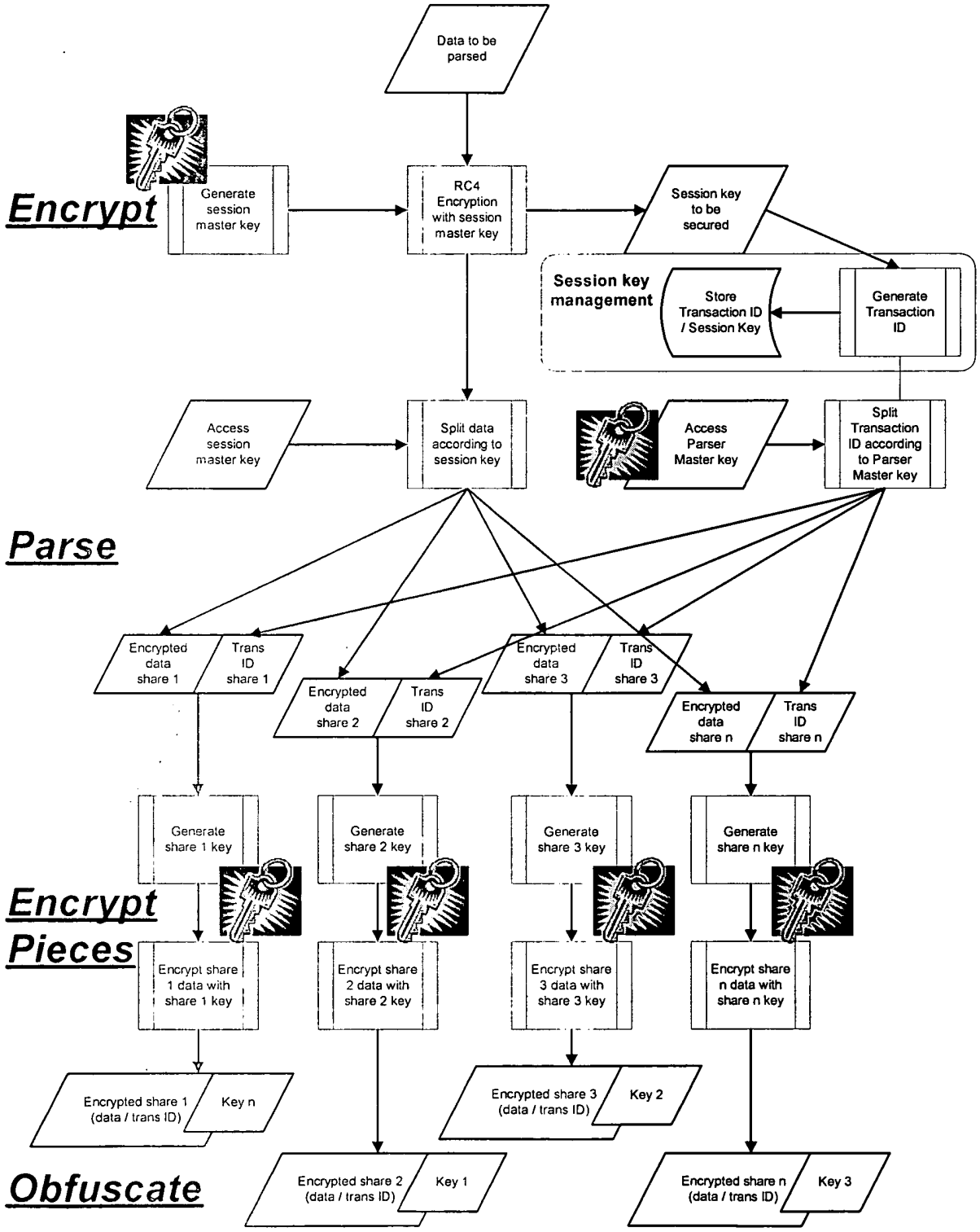

Data to be parsed
Encrypt
Generate session master key
RC4 Encryption with session master key
Session key to be secured
Session key management
Store Transaction ID / Session Key
Generate Transaction ID
Access session master key
Split data according to session key
Access Parser Master key
Split Transaction ID according to Parser Master key
Parse
Encrypted data share 1
Trans ID share 1
Encrypted data share 2
Trans ID share 2
Encrypted data share 3
Trans ID share 3
Encrypted data share n
Trans ID share n
Generate share 1 key
Generate share 2 key
Generate share 3 key
Generate share n key
Encrypt Pieces
Encrypt share 1 data with share 1 key
Encrypt share 2 data with share 2 key
Encrypt share 3 data with share 3 key
Encrypt share n data with share n key
Encrypted share 1 (data / trans ID)
Key n
Encrypted share 3 (data / trans ID)
Key 2
Obfuscate
Encrypted share 2 (data / trans ID)
Key 1
Encrypted share n (data / trans ID)
Key 3

**Figure 23**

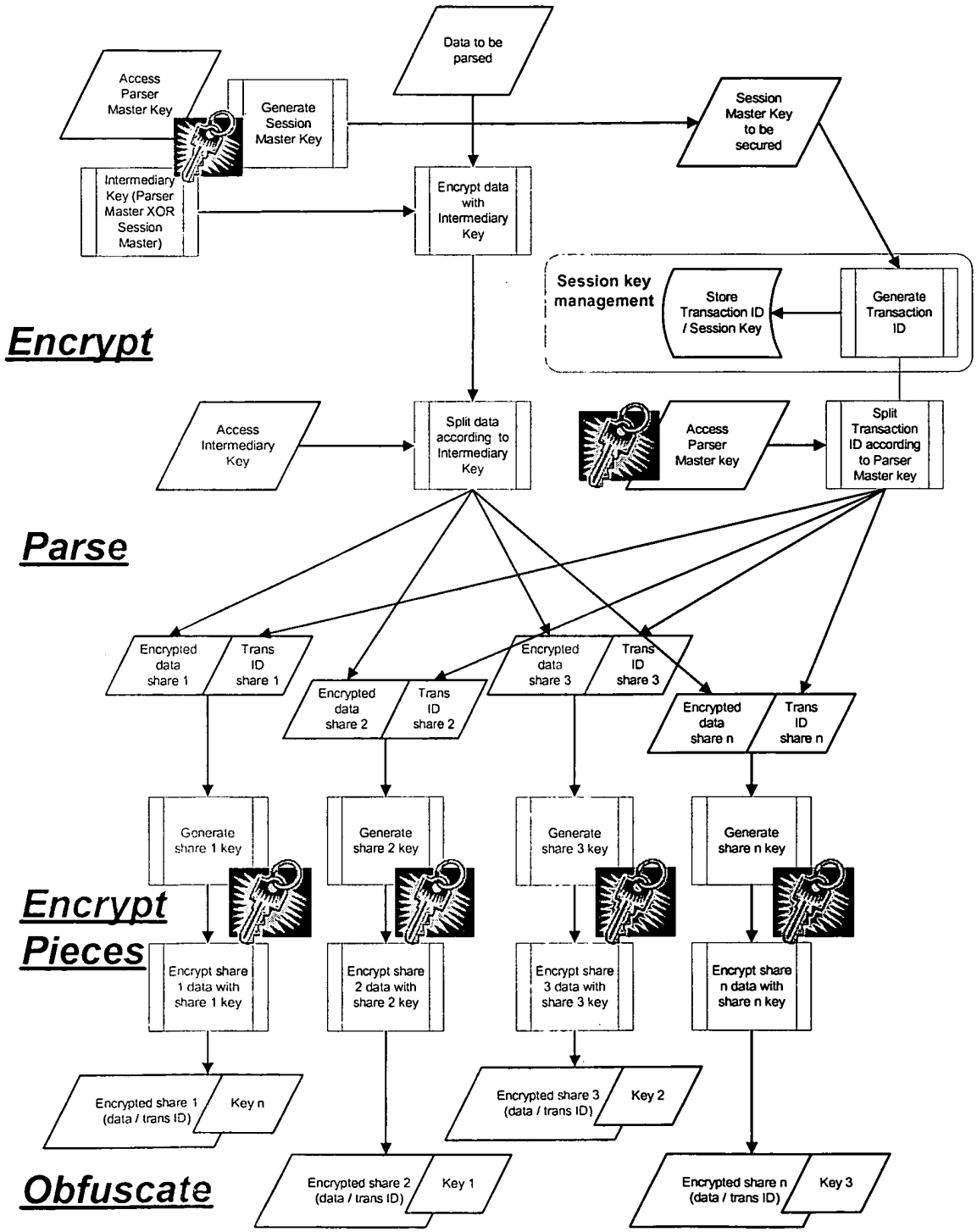

Data to be parsed
Access Parser Master Key
Generate Session Master Key
Session Master Key to be secured
Intermediary Key (Parser Master XOR Session Master)
Encrypt data with Intermediary Key
Session key management
Store Transaction ID / Session Key
Generate Transaction ID
Encrypt
Access Intermediary Key
Split data according to Intermediary Key
Access Parser Master key
Split Transaction ID according to Parser Master key
Parse
Encrypted data share 1
Trans ID share 1
Encrypted data share 2
Trans ID share 2
Encrypted data share 3
Trans ID share 3
Encrypted data share n
Trans ID share n
Generate share 1 key
Generate share 2 key
Generate share 3 key
Generate share n key
Encrypt Pieces
Encrypt share 1 data with share 1 key
Encrypt share 2 data with share 2 key
Encrypt share 3 data with share 3 key
Encrypt share n data with share n key
Encrypted share 1 (data / trans ID)
Key n
Encrypted share 3 (data / trans ID)
Key 2
Obfuscate
Encrypted share 2 (data / trans ID)
Key 1
Encrypted share n (data / trans ID)
Key 3

**Figure 24**

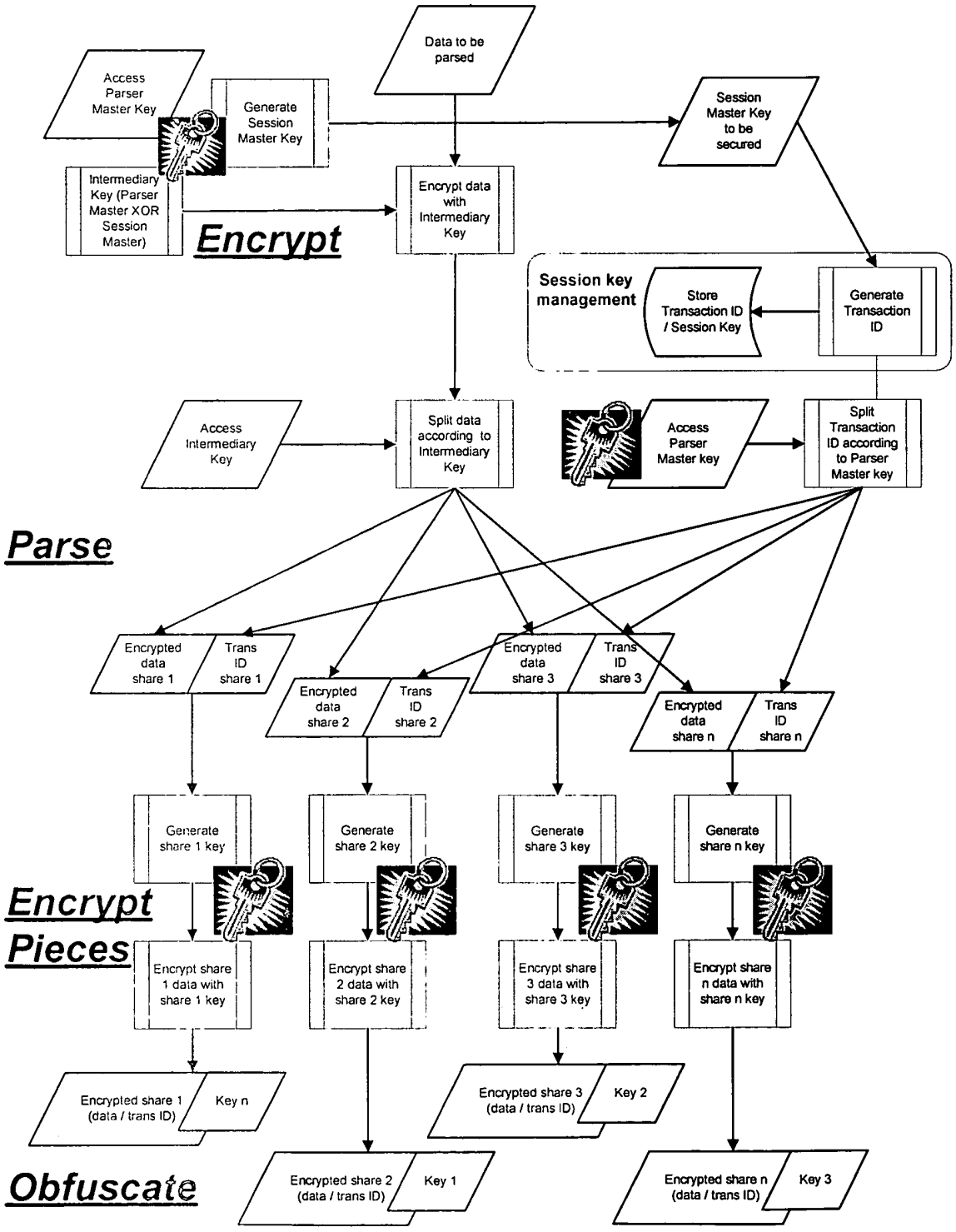
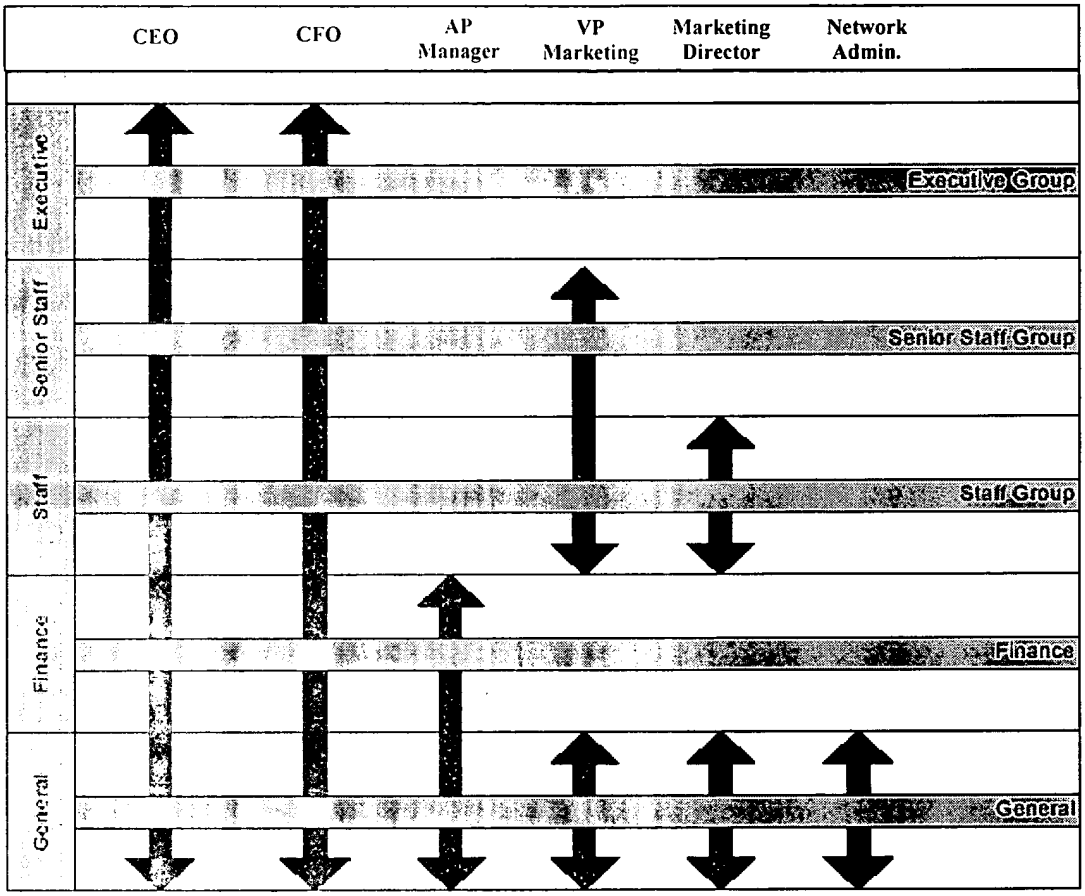

Data to be parsed
Access Parser Master Key
Generate Session Master Key
Session Master Key to be secured
Intermediary Key (Parser Master XOR Session Master)
Encrypt
Encrypt data with Intermediary Key
Session key management
Store Transaction ID / Session Key
Generate Transaction ID
Access Intermediary Key
Split data according to Intermediary Key
Access Parser Master key
Split Transaction ID according to Parser Master key
Parse
Encrypted data share 1
Trans ID share 1
Encrypted data share 2
Trans ID share 2
Encrypted data share 3
Trans ID share 3
Encrypted data share n
Trans ID share n
Generate share 1 key
Generate share 2 key
Generate share 3 key
Generate share n key
Encrypt Pieces
Encrypt share 1 data with share 1 key
Encrypt share 2 data with share 2 key
Encrypt share 3 data with share 3 key
Encrypt share n data with share n key
Encrypted share 1 (data / trans ID)
Key n
Encrypted share 3 (data / trans ID)
Key 2
Obfuscate
Encrypted share 2 (data / trans ID)
Key 1
Encrypted share n (data / trans ID)
Key 3

Figure 25


CEO
CFO
AP Manager
VP Marketing
Marketing Director
Network Admin.
Executive
Executive Group
Senior Staff
Senior Staff Group
Staff
Staff Group
Finance
Finance
General
General

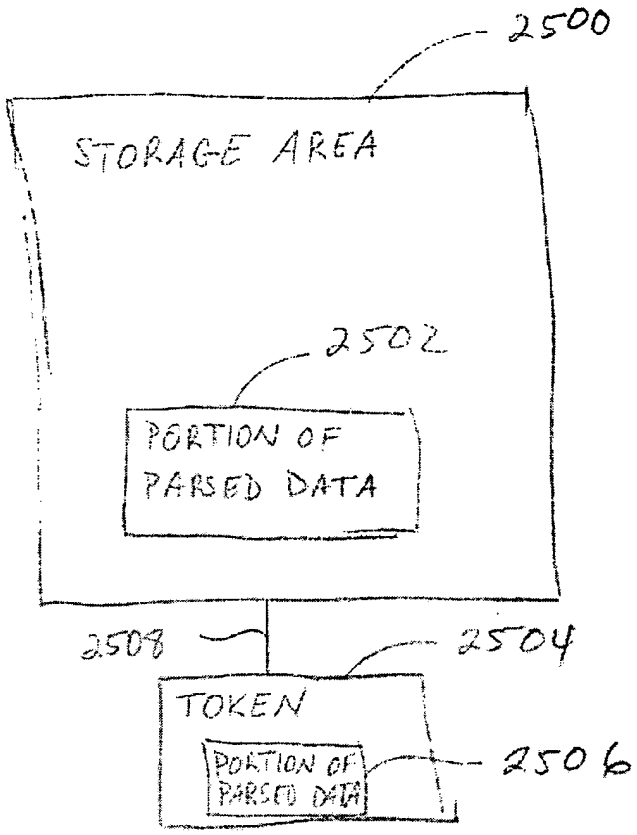

2500
STORAGE AREA
2502
PORTION OF PARSED DATA
2508
2504
TOKEN
PORTION OF PARSED DATA
2506

FIGURE 26

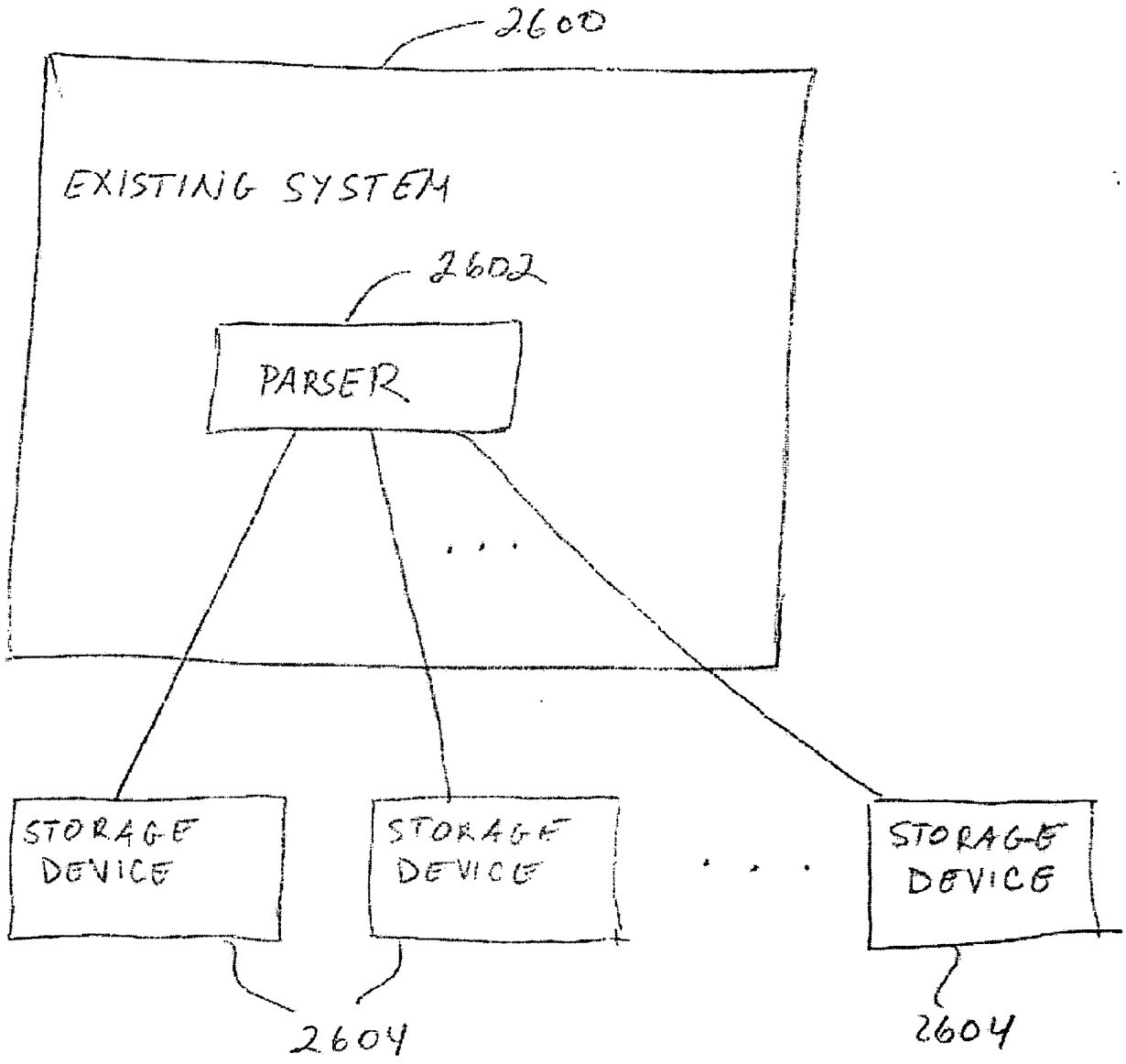

2600
EXISTING SYSTEM
2602
PARSER
STORAGE DEVICE
STORAGE DEVICE
STORAGE DEVICE
2604
2604

FIGURE 27

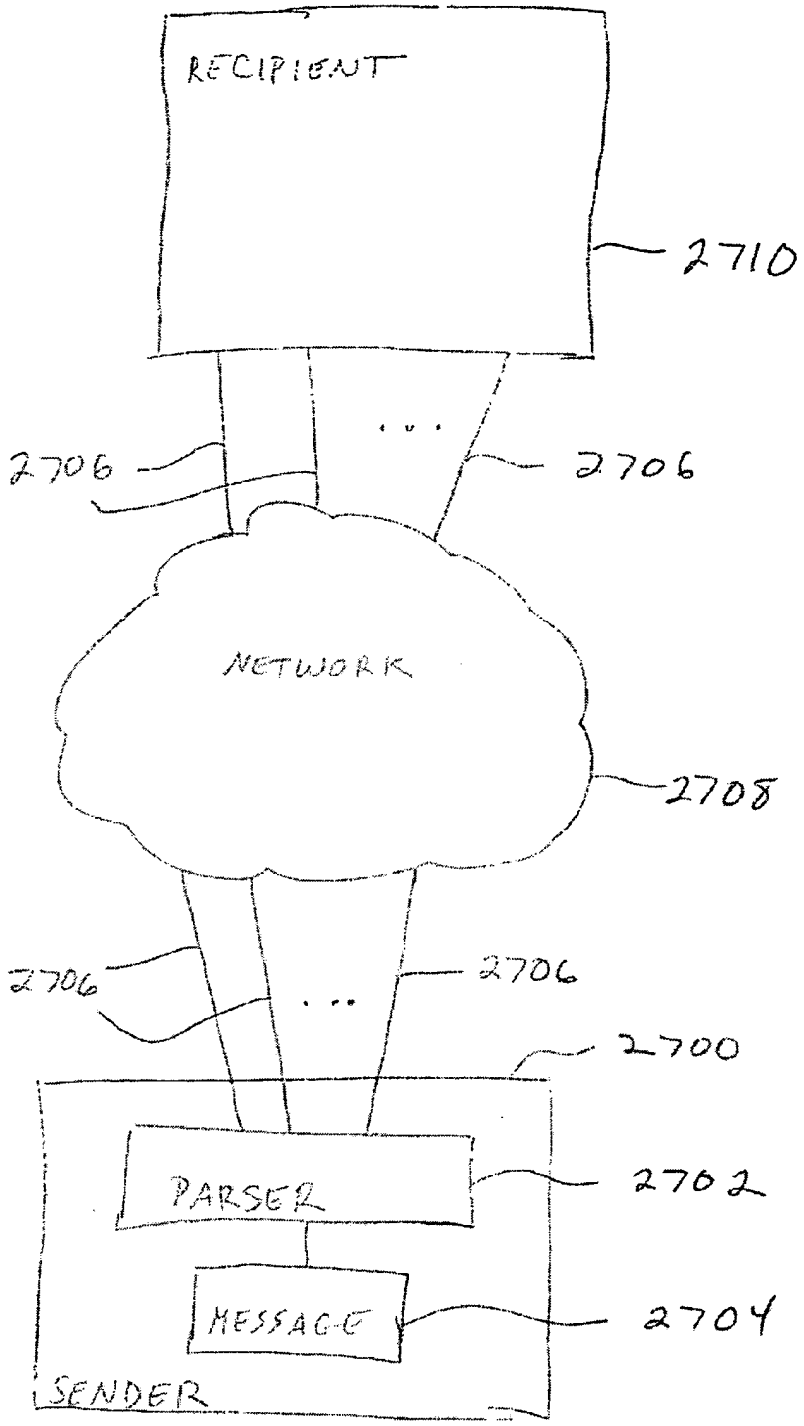

RECIPIENT
2710
2706
2706
NETWORK
2708
2706
2706
2700
PARSER
2702
MESSAGE
2704
SENDER

FIGURE 28

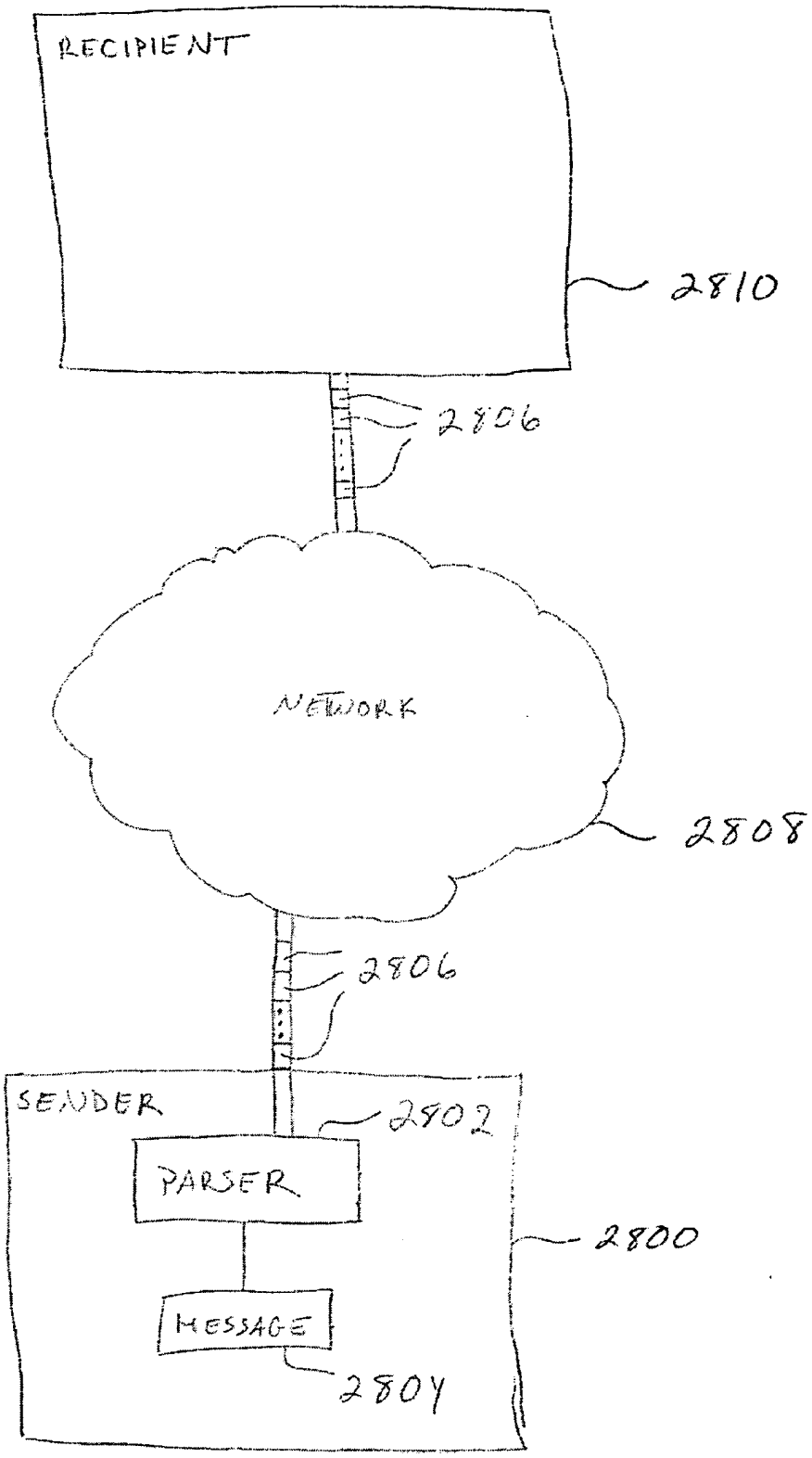
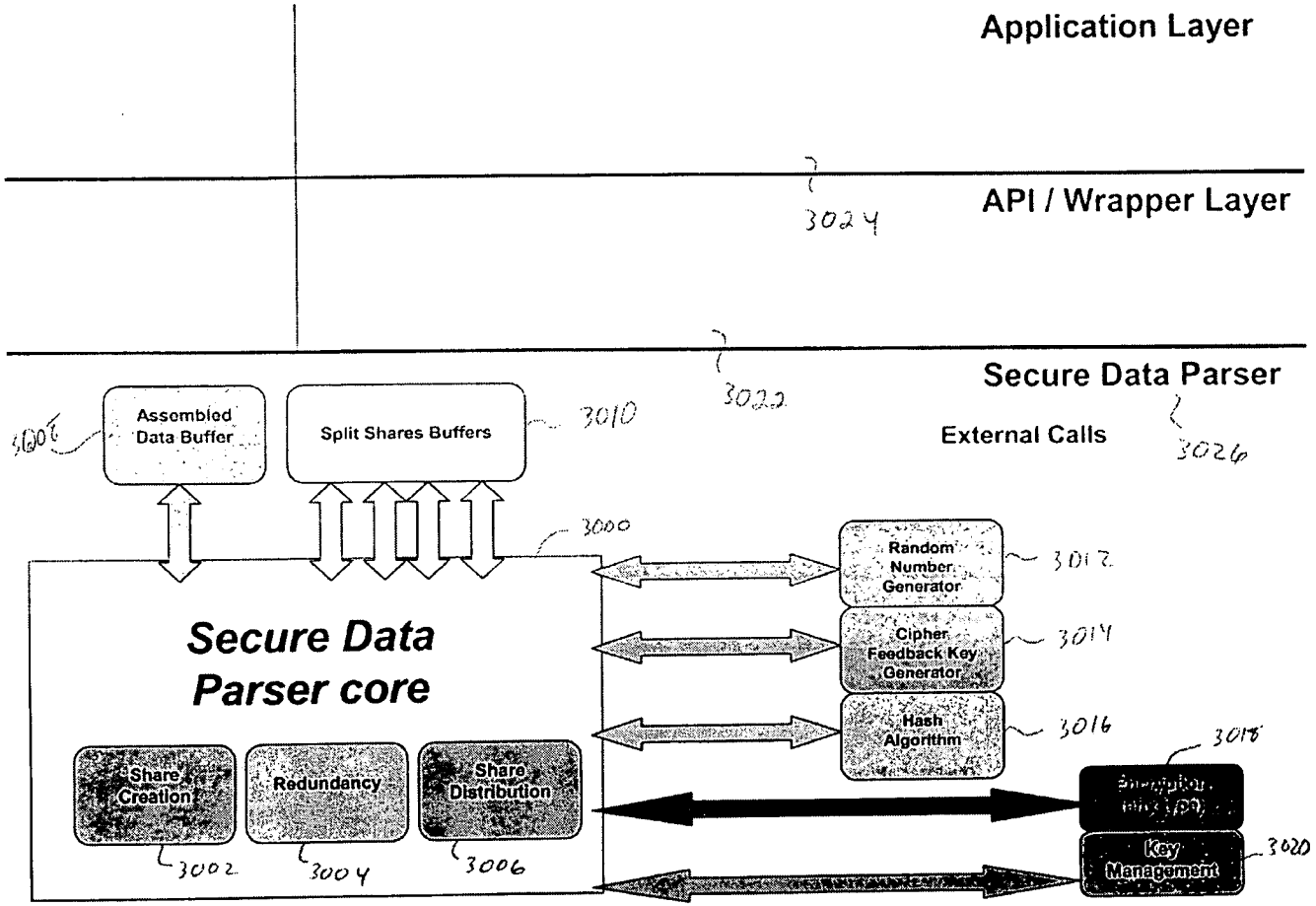

RECIPIENT
2810
2806
NETWORK
2808
2806
SENDER
2802
PARSER
2800
MESSAGE
2804

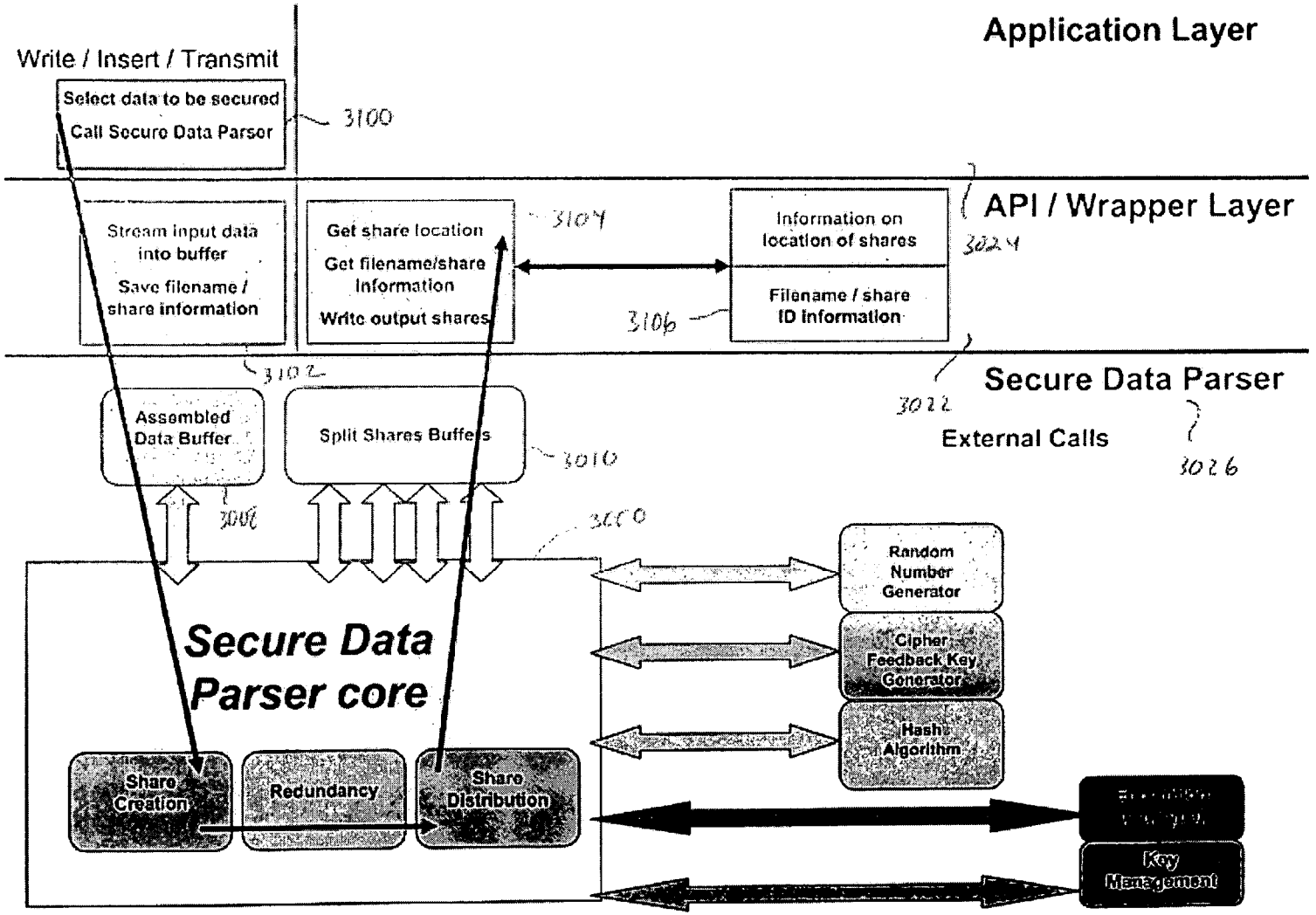
FIGURE 29




Interface to the SecureParser is facilitated by the creation of an API / Wrapper Layer between the Application and the Parser

FIGURE 30

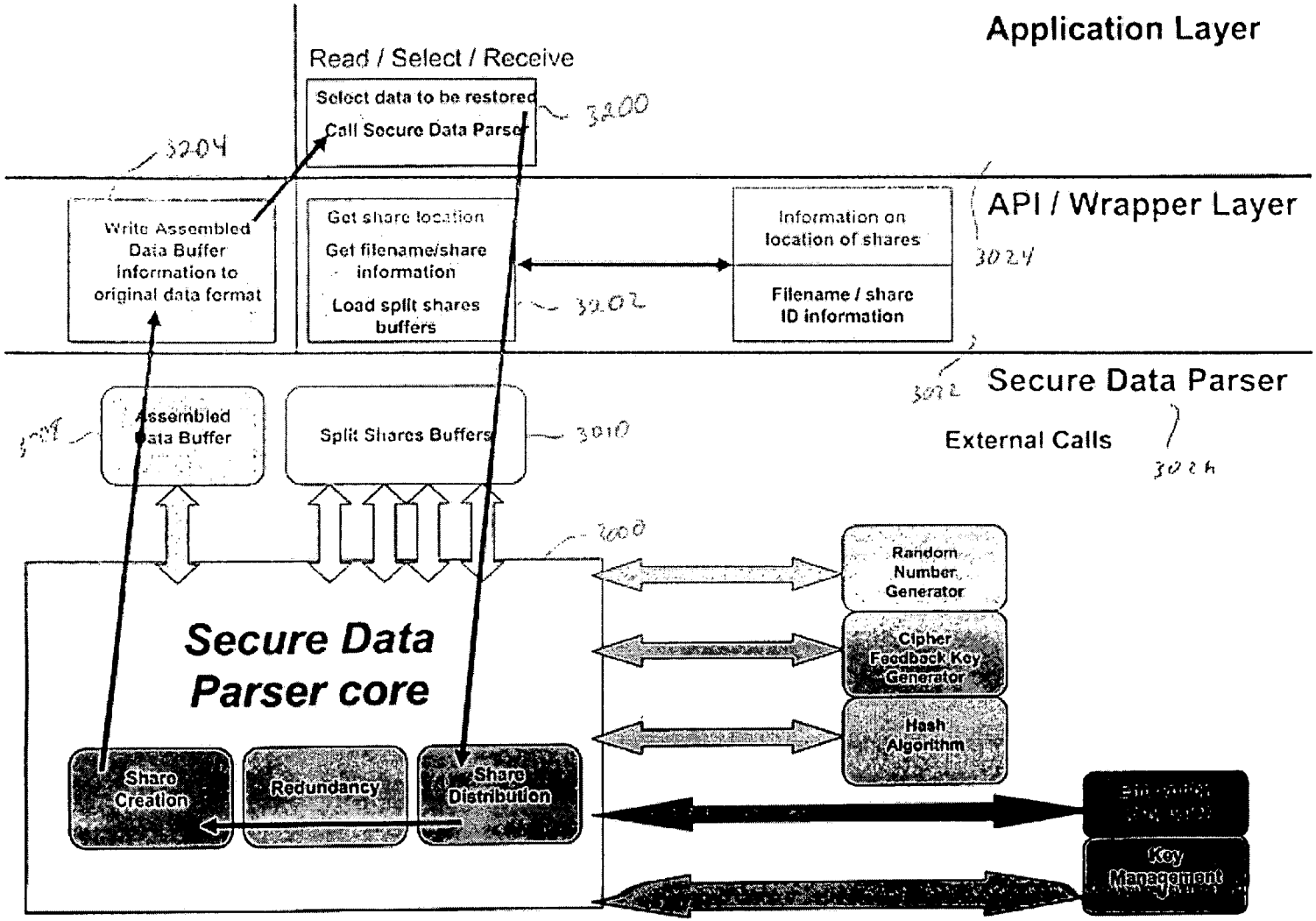
EX1015, Page 366




**The API layer handles streaming of the data to the SecureParser and secure storage of the resulting shares**

FIGURE 3.1

EX1015, Page 367




**For retrieval of information, the host application identifies the data to be restored and invokes the Secure Data Parser API**

FIGURE 32

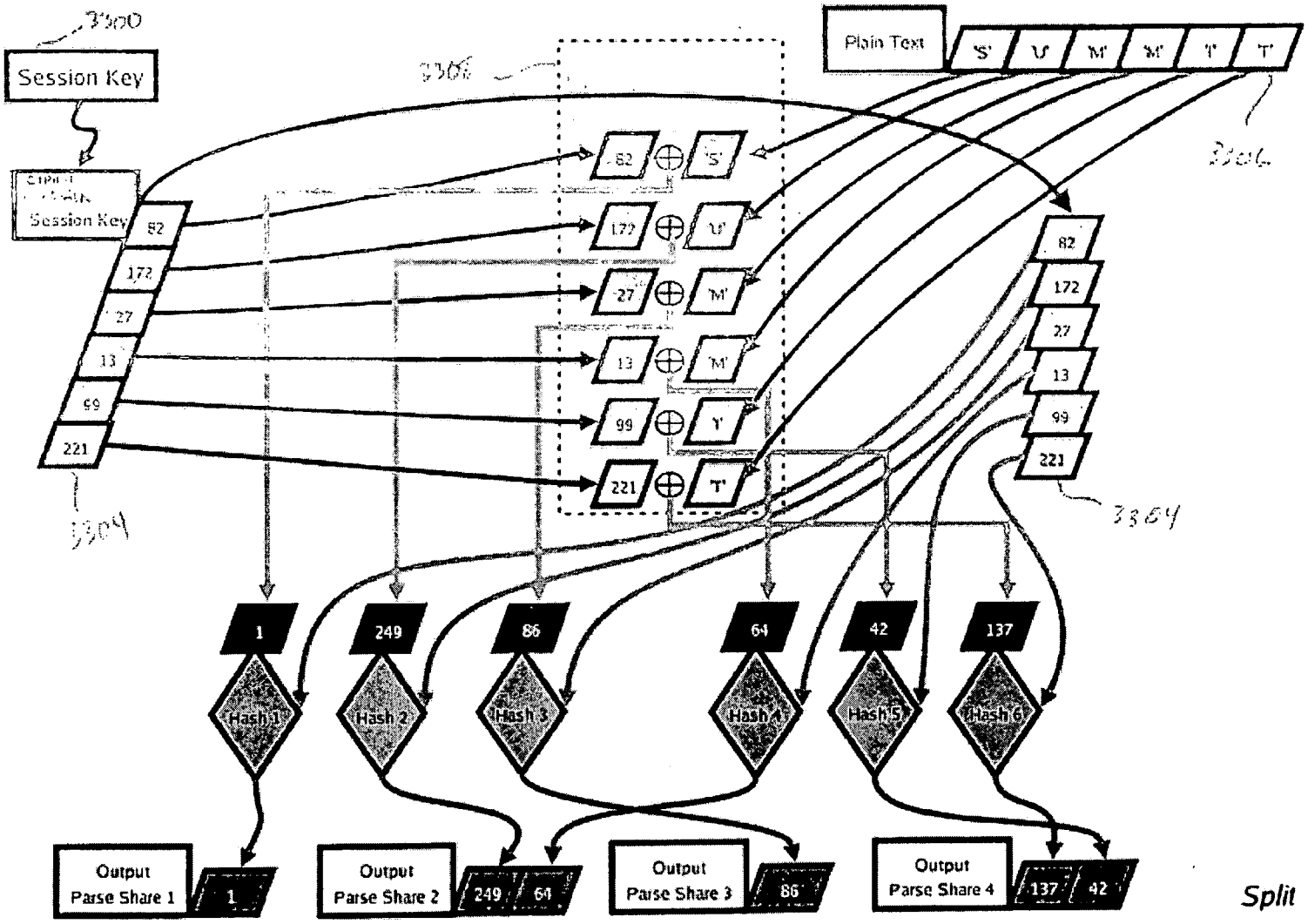



FIGURE 33

EX1015, Page 369

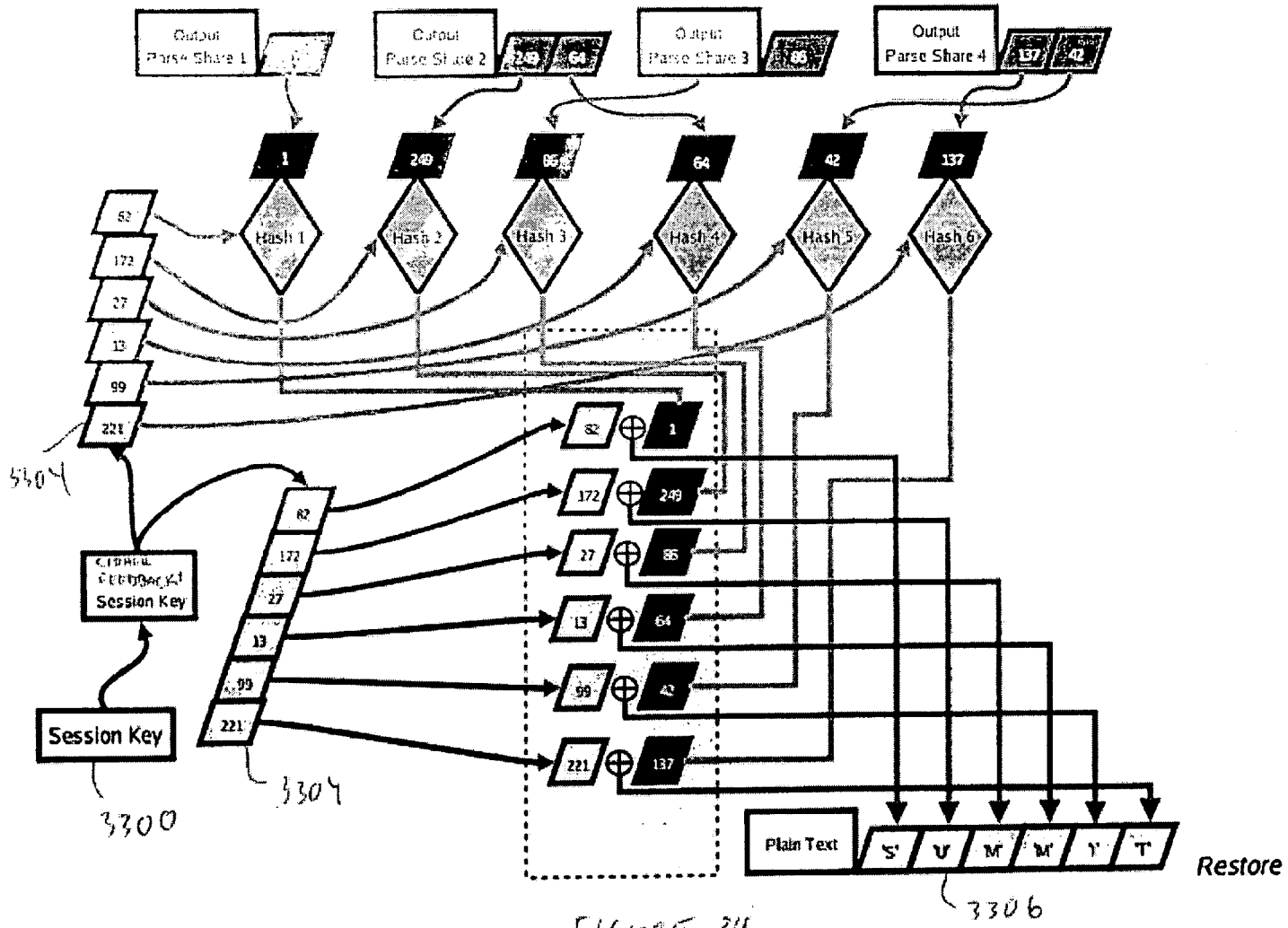



FIGURE 34

EX1015, Page 370

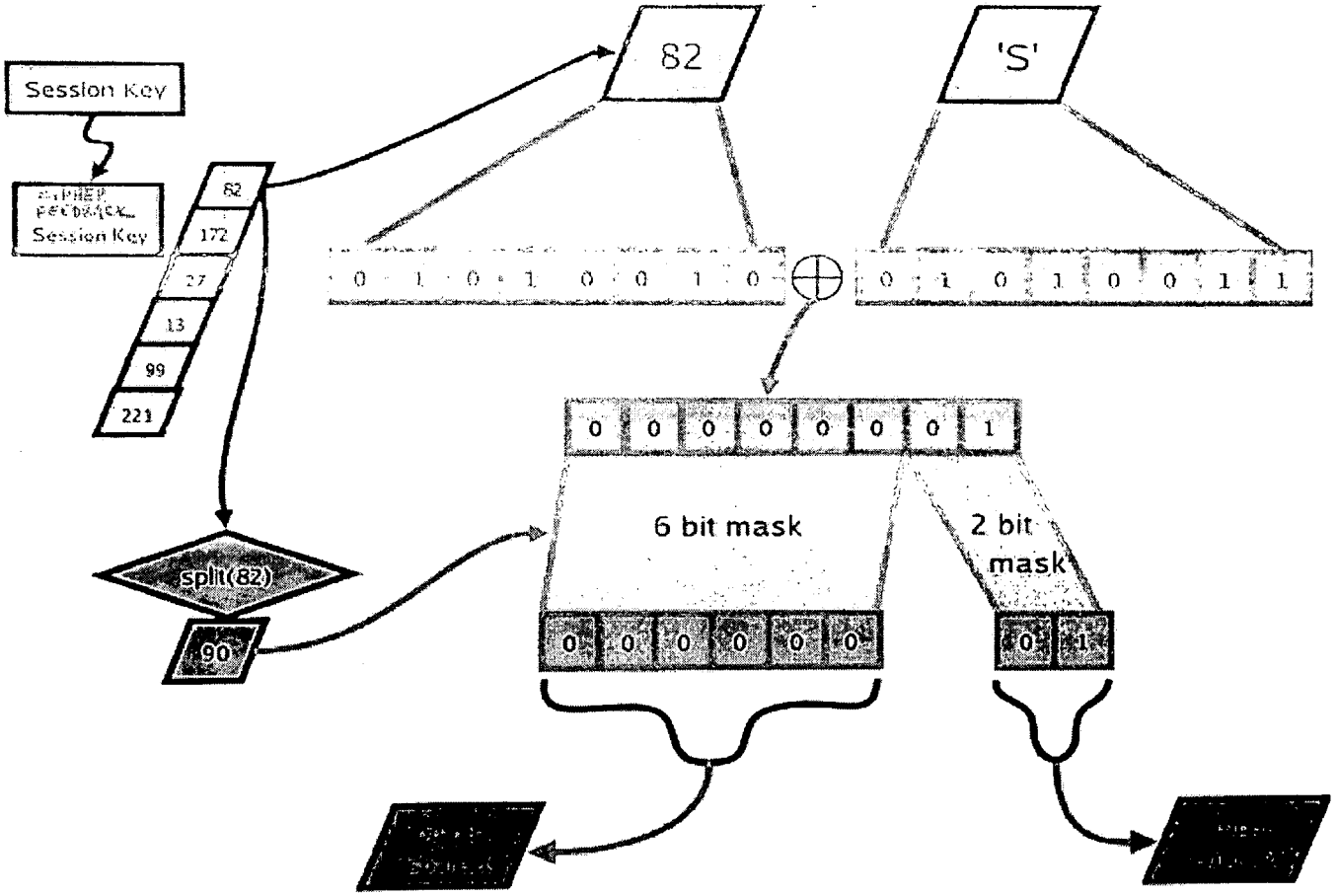

Session Key
Session Key
82
172
27
13
99
221
split(82)
90
82
'S'
0 1 0 1 0 0 1 0
0 1 0 1 0 0 1 1
0 0 0 0 0 0 0 1
6 bit mask
2 bit mask
0 0 0 0 0 0
0 1

FIGURE 35

# This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

## BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☒ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☒ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

**IMAGES ARE BEST AVAILABLE COPY.**
**As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.**

PATENT APPLICATION SERIAL NO _____

U.S. DEPARTMENT OF COMMERCE
PATENT AND TRADEMARK OFFICE
FEE RECORD SHEET

09/21/2005 FMETEKI1 00000117 061075 60718185

01 FC:2005 100.00 DA
02 FC:2085 750.00 DA

PTO-1556
(5/87)

U.S. Government Printing Office: 2002 — 466-247/69033

INVENTOR INFORMATION

Inventor One Given Name:: Rick
Family Name:: Orsini
Postal Address Line One:: 2100 KIngs Forest Lane
City:: Flower Mound
State or Province:: TX
Country:: United States
Postal or Zip Code:: 75028
City of Residence:: Flower Mound
State or Province of Residence:: TX
Country of Residence:: United States
Citizenship Country:: United States
Inventor Two Given Name:: Mark S
Family Name:: O'Hare
Postal Address Line One:: 8 Kennedy Court
City:: Coto De Caza
State or Province:: CA
Country:: United States
Postal or Zip Code:: 92679
City of Residence:: Coto De Caza
State or Province of Residence:: CA
Country of Residence:: United States
Citizenship Country:: United States
Inventor Three Given Name:: Roger
Family Name:: Davenport
Postal Address Line One:: 3204 CR 4119
City:: Campbell
State or Province:: TX
Country:: United States
Postal or Zip Code:: 75422
City of Residence:: Campbell
State or Province of Residence:: TX
Country of Residence:: United States
Citizenship Country:: 75422
Inventor Four Given Name:: Paras
Family Name:: Traganos
Postal Address Line One:: 21 Ewell Avenue
City:: Lexington
State or Province:: MA
Country:: United States
Postal or Zip Code:: 02421
City of Residence:: Lexington
State or Province of Residence:: MA
Country of Residence:: United States
Citizenship Country:: United States
Inventor Five Given Name:: Steve
Family Name:: Winick
Postal Address Line One:: 3 Expressway Plaza
City:: Roslyn Heights

State or Province:: NY
Country:: United States
Postal or Zip Code:: 11577
City of Residence:: Roslyn Heights
State or Province of Residence:: NY
Country of Residence:: United States
Citizenship Country:: United States

CORRESPONDENCE INFORMATION

Correspondence Customer Number:: 1473
Fax One:: 2125969090
Electronic Mail One:: ashvarts@fishneave.com

APPLICATION INFORMATION

Title Line One:: SECURE DATA PARSER METHOD AND SYSTEM
Total Drawing Sheets:: 36
Formal Drawings?:: No
Application Type:: Provisional
Docket Number:: SEC-3 Prov
Secrecy Order in Parent Appl.?:: No

REPRESENTATIVE INFORMATION

Representative Customer Number:: 1473
Registration Number One:: 47943
Registration Number Two:: 31069

Source:: PrintEFS Version 1.0.1