VenusIDS: An Active Database Component for Intrusion Detection*

Lane B. Warshaw, Lance Obermeyer, Daniel P. Miranker, Sara P. Matzner
Applied Research Laboratories / Department of Computer Sciences,
The University of Texas at Austin
{warshaw, lanceo, miranker, matzner}@arlut.utexas.edu

Abstract

Active-databases are a budding technology where rule-based expert systems can be developed in tight integration with database management systems. This paper presents VenusIDS: an active database component of the Network Exploitation Detection Analyst Assistant (NEDAA) developed as an enhancement to the analysis layer of a two-layer distributed network intrusion detection system using the VenusDB active database system. The layers consist of a network layer and an analysis layer. The network layer contains probes on each subnetwork that sniff network traffic and forward interesting packets in real time to a central Oracle database. The analysis layer comprises this central database and the mechanism to identify and report intrusions.

For active-database technology to form an effective basis for intrusion detection, it must be capable of processing network events at least as fast as the network probes produce and log them. Our performance results show that VenusIDS is more than fast enough to handle this rate. Further, VenusIDS is scalable in the number of rules and size of the underlying database. As context for the VenusIDS component, we begin by describing the application architecture and the VenusDB system, with emphasis on the particular features that are important to distributed intrusion detection. We follow that with a description of the VenusIDS component and its performance profile that enables near real time intrusion detection. We conclude with a discussion of future topics for active-database analysis layers.

* This work was funded under Contracts N00039-D-0051,Task Order No. 0293 and Task Order No. 0273.

PAN EX1008
Palo Alto Networks v. BT Americas
IPR2023-00889

VenusIDS: An Active Database Component for Intrusion Detection

Abstract

Active-databases are a budding technology where rule-based expert systems can be developed in tight integration with database management systems. This paper presents VenusIDS: an active database component of the Network Exploitation Detection Analyst Assistant (NEDAA) developed as an enhancement to the analysis layer of a two-layer distributed network intrusion detection system using the VenusDB active database system. The layers consist of a network layer and an analysis layer. The network layer contains probes on each subnetwork that sniff network traffic and forward interesting packets in real time to a central Oracle database. The analysis layer comprises this central database and the mechanism to identify and report intrusions.

For active-database technology to form an effective basis for intrusion detection, it must be capable of processing network events at least as fast as the network probes produce and log them. Our performance results show that VenusIDS is more than fast enough to handle this rate. Further, VenusIDS is scalable in the number of rules and size of the underlying database. As context for the VenusIDS component, we begin by describing the application architecture and the VenusDB system, with emphasis on the particular features that are important to distributed intrusion detection. We follow that with a description of the VenusIDS component and its performance profile that enables near real time intrusion detection. We conclude with a discussion of future topics for active-database analysis layers.

1 Introduction

We have applied active database technology to enhance a two-layered network intrusion detection system (IDS) with near real time capabilities. The two layers of this IDS consist of a network layer and an analysis layer. The network layer contains an array of protected sub-networks. A privileged computer on each sub-network, called a *probe*, sniffs all network traffic. Probes execute an initial set of filters that analyzes traffic, capturing suspicious packets while discarding obvious non-threatening connections. A synopsis of the results and the filtered subset of connections are forwarded to the analysis layer and logged to an Oracle database. At the analysis layer, an ad-hoc C program with embedded SQL periodically wakes and further analyzes the activities. This analysis searches for possible intrusion patterns and raises alarms for the human network analysts if suspicious activity is detected. If further action is required, the network analyst may exploit an array of database query tools to investigate and track an individual alarm event.

VenusIDS (The VenusDB Intrusion Detection System) is a component of the Network Exploitation Detection Analyst Assistant (NEDAA), an intrusion detection system that provides a supportive environment for human intelligence, developed by the Applied Research Laboratories; The University of Texas at Austin (ARL:UT). The VenusIDS component better automates the analysis layer through the use of active database technology. Active database technology is employed to encode the possible intrusion patterns directly in declarative rules that execute in tight integration with the **existing** database in near real time. This has the advantage of significantly reducing the latency between the completion of a possible intrusion pattern and the notification of the human network analyst. Further, our experience in other domains demonstrates that encoding the intrusion detection patterns into declarative rules will make the rules more extensible, more understandable, and less error prone [5,15].

The many ways of generating the filtering rules used by the network probes and the rules used by the analysis layer are of secondary concern in this paper. For example, the rule set used by NEDAA contains both rules automatically generated by machine learning algorithms as well as human created domain knowledge rules. In another application [13], given a set of intrusion patterns, the NetSTAT

system can generate a set of probe filtering rules. Data mining techniques have been employed for rule generation and for subsequent monitoring at the analysis layer [10]. This paper is not concerned with rule generation, instead we focus on presenting the VenusIDS as a tool that is sufficiently declarative and efficient to support any of these techniques.

VenusIDS is implemented using the VenusDB active database system, an extension of Venus, a family of C++ embedded rule-languages [1,2,6,8]. VenusDB extends the traditional expert system model with facilities for inferencing directly on heterogeneous database tables and monitoring events through exploiting database triggers. This paper demonstrates how the coupling of VenusIDS with the VenusDB active database system results in an effective tool for building sophisticated near real time network intrusion detection systems.

2 Application

VenusIDS is an enhancement of the analysis layer of a two-layer distributed network intrusion detection system. The layers are composed of a network layer and an analysis layer (Figure 1). The network layer consists of an array of subnetworks each monitored by a probe. A probe sniffs, in real time, all network traffic on a subnetwork and filters the traffic by discarding obvious non-threatening traffic and classifying the remaining traffic as suspicious or uncertain. Suspicious traffic, also called *events*, is logged in real-time to the analysis layer, while all non-discarded traffic is logged at predetermined intervals to the analysis layer. In this paper, we do not discuss the implementation of the network probes; we assume that the probes can be configured, automatically or manually, to recognize events needed by the analysis layer.

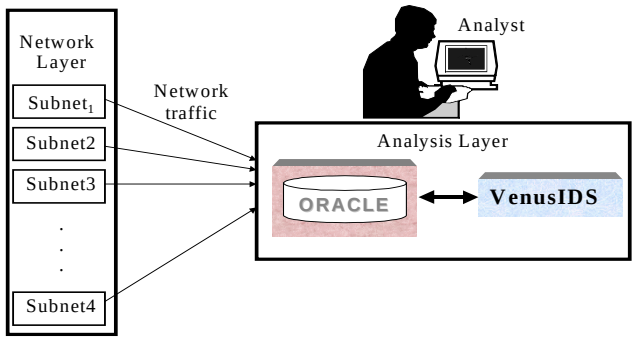

Figure 1. Application Architecture

The analysis layer contains a variety of tools to assist network analysts in decision support. The data in the analysis layer is stored in an Oracle database due to its durable and persistent data storage that can be efficiently searched using Oracle's query services. The network analyst uses the real time events logged from probes to root searches for anomalies, possibly executing queries against the Oracle database. When an anomaly is discovered, the analyst executes the appropriate disciplinary action and produces an anomaly transcript. During this process, analysts often discover English statements describing anomaly patterns. An abstracted example of an English-stated anomaly rule is illustrated in Figure 2.

In the first generation of the analysis layer, the English statements discovered by network analyst were encoded as SQL queries to be run in batch mode at predetermined intervals. This method suffered

```
Event: first connection from a server to an unprotected computer

Rule Name: DNS overflow

Rule Condition: A new X window connection is made for the first time from
                a protected server to an unprotected computer shortly
                after the unprotected computer connected on the server's
                DNS port.
```

Figure 2. Abstracted English Statement of an Anomaly Rule

due to its inherent non-real time execution. Additionally, this method is non-scalable since at every execution, every anomaly discovered by a query will be reported, repeating the work of previous executions and bombarding the analyst with too much information. Figure 2 illustrates this non-scalability; without clever ad-hoc mechanisms, an SQL query encoding the DNS overflow rule will retrieve every DNS overflow in the database each time the query is executed.

VenusIDS is the second generation of the analysis layer using the VenusDB active database system. As stated by Widom and Ceri, "*active database systems enhance traditional database functionality with powerful rule processing capabilities […]*[12]." Thus, VenusIDS exploits active database technology to encode the English statements of anomalies as expert system rules that are evaluated in real time from the occurrence of events discovered by network probes. In Sections 3.1 and 4.2 we will explain how the rules, such as the one illustrated in Figure 2, are straightforwardly encoded within VenusIDS.

We begin with a discussion of VenusDB giving special mention to system features that are directly utilized by VenusIDS. We then present the VenusIDS architecture followed by preliminary performance results. We conclude with a discussion of ongoing and future work.

3 VenusDB

VenusDB comprises a family of rule systems. The foundation system, referred to as Venus, is a main-memory expert system environment [1]. Venus includes a general-purpose rule language embedded within C++, an optimizing compiler, and a highly efficient match algorithm that yields fast scalable execution speed [6]. Additionally, the Venus language encourages structured programming through a method of parameterized rule modules. A case study comparing Venus and another classic expert system language demonstrated dramatic improvements in four quantitative measures of code quality, suggesting an improvement in programmer productivity by using Venus [14].

3.1 Language

Venus rules are organized into parameterized groups called modules. See Figure 3. Modules are designated by the keyword `module` followed by a list of formal parameters and local variables. The formal parameters and the local variable list are made up of containers and primitive variables. A container is Venus' set data type used for rule inference. Containers are distinguished by the use of square brackets "[ ]", and their elements are defined as C++ class instances. Primitive variables are single inferable objects.

A Venus rule is basically a single legal C++ `if` statement that contains three parts: a *header*, a *guard*, and an *action*. The header contains the keyword `rule` followed by a rule name, an optional priority and a declaration section. The declaration section begins with a `from` clause. The `from` clause, borrowed from SQL, associates aliases with cursor variables that quantify Venus containers. A cursor can be quantified either existentially or universally. An existentially quantified cursor is denoted by a "?" within the square brackets. A universally quantified cursor is denoted by a "*" within the square brackets.

VenusDB extends the Venus declaration section with an optional event clause. The event clause implements e*vent-condition-action* (ECA) rules commonly found in active database systems [12]. ECA rules offer the programmer greater flexibility to manage the control search by restricting the applicability of a rule to a particular events. When an event clause is not listed, the specification defaults to all events and execution proceeds equivalently to Venus. The event clause begins with the keyword `event`, followed by the event to trigger on and the object that is being monitored.

The guard, (or `if` part) is a restricted C++ logical expression. It may refer to local variables, actual parameters, cursor aliases and function/method calls in the standard way.

The action is a restricted list of C++ expressions. Expressions include function calls, assignment statements, and Venus module calls. The compiler parses the action and automatically recognizes updates to data elements, appropriately notifying the Venus inference engine.

```
module monitorConnectionEvents(CONNECTIONEVENTS connection,
                               CONNECTIONS Connections[],
                               CONNECTIONEVENTS SeenConnections[]) {

   rule dnsoverflow; /* rule to test for a possible DNS overflow */
   from Connections[?] prev_connections;    // existential cursor
        SeenConnections[*] oldconnections;  // universal cursor
   event none SeenConnections;              // event clause
   if(connection != oldconnections  && /* never have seen this connection */
      connection.destPort == XWINPORT   && /* XWindows */

      /* computer previously connected to the server */
      connection.destIp == prev_connection.sourceIp &&
      connection.sourceIp == prev_connection.destIp &&

      prev_connection.destPort == DNSPORT  && /* DNS port */

      /* previous connection was recent */
      prev_connection.endTime – connection.startTime <=2) {

         signalAlert(connection,prev_connection); /* call alert handler */
} }
```

Figure 3. Sample Venus module and rule

For example, consider Figure 3, the encoding of the DNS overflow rule described in Figure 2. The `monitorConnectionEvents` module contains only one rule, the `dnsoverflow` rule. The module's formal parameter list contains the primitive variable `connection` and the containers `Connections`, containing all the connections ever seen by the bank of subnetworks, and `SeenConnections`, containing a non-repeating list of previously seen connections. The `from` clause creates an existentially quantified cursor over the `Connections` container, `prev_connection`, and the universally quantified cursor over the `SeenConnections` container, `oldconnections`. Next, an event clause is listed turning off events on the `SeenConnections` container for the `dnsoverflow` rule, in effect, precluding VenusDB from evaluating the `dnsoverflow` rule on modifications to the `SeenConnections` container. The rule's guard is described in Figure 2.

3.2 Abstract Machine Interface (AMI)

VenusDB operates by issuing commands to the abstract machine interface (AMI), an instruction set used by Venus's runtime match algorithm. The purpose of the AMI is to tightly integrate expert system and database behavior and to provide for heterogeneous data components. This integration is accomplished by encapsulating all database functionality behind a uniform API (abstract programming interface) [8]. The AMI is the only system to publish such a high-level interface between the rule and database portions.

The AMI is defined by a set of abstract C++ classes. As a consequence of this object-oriented definition, no special syntax is used to differentiate main memory containers, object oriented or relational database tables, or unstructured data sources within VenusDB rule code. For example, again refer to Figure 3. In the `dnsoverflow` rule, the `Connections` container may store its data in an Oracle database while the `SeenConnections` container may store its data in a Microsoft Access database. The storage structure of a container is only determined by the container's instance declaration[1].

3.3 Layered Active Databases

A *layered* active database system treats the database as a black box in which the only access to the database is through public interfaces. An advantage of this architecture is that the database internals

[1] Though the declaration of instance containers is simple due to the use of object-oriented technology, we omit a thorough discussion due to space limitations [8].

need not be available. Thus, the porting and use of heterogeneous components in a layered system requires minimal effort. For this reason, the layered approach is the only choice for VenusDB and its AMI access to component data sources.

A layered approach has proven extremely useful to VenusIDS. Through layering, VenusIDS can easily adopt to the numerous and ever-changing modern intrusion detection probes. Once the appropriate AMI instances have been defined, the addition or modification of a probe does not require any special knowledge for updating. Instead, only changes to the object wrapper code, not the underlying system, will need to be performed.

3.4 VenusDB optimizations

The VenusDB rules in an application may be considered statements upon a *federated database schema* – a single representation of a heterogeneous database that unifies the schemas of all the component data sources. VenusDB, like most loosely coupled federated database systems, decomposes a statement upon the federated schema into one or more statements upon the component schemas and then recomposes the result [11].

VenusDB is designed to leverage the fast-access data retrieval methods of its component data sources including the use of indexing and advanced query support when available. This occurs through pushing predicates down to the local database level. The VenusDB compiler identifies predicates from rule guards that are applicable to be executed on a component database. These predicates are then passed through to the local database for execution by the local database's query facility.

This scheme results in a significant performance benefit for VenusIDS. Many of VenusIDS's rules test very large tables within its component Oracle database[2]. Without the predicate pushdown facility, a rule investigating these large tables would require entire tables to be retrieved over the network and tested against the rule guard. With the predicate pushdown facility, only the data that satisfies the rule guard is transferred over the network.

VenusDB's predicate pushdown facility directly correlates to the goals of intrusion detection systems. Such goals include presenting analysts with the most detailed and useful information possible through the reduction of false positives and recognition of the maximal set of *real* intrusions. This implies that a well-written intrusion detection rule should be satisfied a minimal number of times since, in most cases, a rule satisfaction indicates the discovery of an alert. With the predicate pushdown facility, VenusDB will push this predicate to the component database and, due to the goals of intrusion detection systems, only a minimal amount of data from the database will satisfy the predicate. Thus, predicate pushdown forces VenusIDS to execute most of the work within the advanced query facilities of the component database, and additionally, eliminates the network overhead of transferring large amounts of data.

3.5 Venus Modularity and Semantics

A Venus rule is said to *fire* and execute its action when the rule is selected to be evaluated and its guard is satisfied (evaluated to "true"). The entire action of a Venus rule is defined to be a single atomic transition in a state-space (a transaction) and rules fire by a fair non-deterministic policy. Venus, being a general-purpose rule language, provides for *rule chaining* – the action of a rule may change the state of the system and cause more rules to be evaluated and fire due to this change. When the firing of rules no longer modify the current state or the modified state satisfies no rules, a Venus module is said to have reached *fixed point* and the module concludes execution [4].

Modules may be listed in the action of a rule and can be nested arbitrarily deeply. If a rule fires and its action lists module calls, then the rules within the nested modules must achieve fixed point before the action of the rule commits. Thus, Venus semantics and nested transaction models are closely related. The resulting execution corresponds to a depth-first traversal of the *module-call graph*, and though declaratively defined, is consistent with procedural intuition.

[2] The connections log table in VenusIDS averages over 11 million rows.

4 Application Architecture

VenusIDS is an active-database intrusion detection system layered upon an Oracle database[3]. Due to the VenusDB's layered architecture, VenusIDS is implemented without any modifications to the existing database application.

4.1 Events

VenusIDS is designed to react to the events discussed in Section 2. We present some basic definitions to formally describe events within an active database framework.

Borrowing from the terminology of Prolog, we define an *intensional* container as a container that maps over a table that is visible only within the rule system, these are working containers for the rule system. Any non-intensional container is an *extensional* container [11].

We distinguish between *monitored* containers and *polled* containers. A monitored container is any container where the rule component is automatically notified of any changes to the underlying table. A polled container is one where a stream of updates is not available to the rule system.

VenusIDS rules chain only due to modifications to monitored containers. Changes to extensional monitored containers are referred to as events, and VenusIDS rule sets are triggered for evaluation only when events occur.

4.2 Module Structure – Rule Base Architecture

VenusIDS uses Venus's module facility to enhance system structure and implement its event driven property by containing a Venus module for each event on an extensional monitored container. Each module's parameter list is composed of a primitive variable from one of the extensional monitored containers followed by the list of the extensional and intensional containers used by VenusIDS rules. The primitive variable within each module contains the row within the extensional container that was modified when spawning an event, and an event triggers evaluation only to its corresponding event module. Due to Venus' semantics, we call the processing of an event from the time of the Venus module invocation until it reaches fixed point as a *transaction*.

For example, Figure 3 illustrates the module that looks for intrusions based on changes to the `CONNECTIONEVENTS` table in the Oracle database. Consider an event consisting of the first time a connection is made between two computers. The connection will spawn an event, wake the `monitorConnectionEvents` module and be passed as the primitive variable `connection`. The module will then begin its search to determine if `connection` is an anomaly. When the `monitorConnectionEvents` module reaches fixed point, the event's transaction completes.

This modular structure segregates rules based on events, and further, encourages analysts to write rules that directly correspond to the types of rules presented in Figure 2. We believe this to be a valuable tool for improving maintenance and rule development within VenusIDS [14].

4.3 Data Flow

Figure 4 illustrates the data flow of VenusIDS. Data enter the system through a bank of monitored networks and are inserted into an Oracle database. Data are then sent by way of Oracle triggers to an SQL trigger action. The trigger code uses a special programming escape mechanism to send the data to a C program[4]. Next, the C program passes the data through an AF_UNIX socket to the VenusIDS component where the data are inserted into a FIFO queue.

---

[3] Currently, VenusIDS is layered only upon an Oracle database. AMI implementations for Sybase and ObjectStore have also been developed [8]. However, our application has not called for the use of these AMI's.

[4] Currently, it is beyond the SQL standard to regulate program language escapes. We believe this functionality should be covered. If such a standard existed, the trigger code unit would not be necessary thereby eliminating a copy of the inserted row.

Data enter the VenusIDS rule-system by first removing the data from the FIFO queue and then calling the appropriate Venus module by exploiting a C++ virtual function call. The rule component then may remove, modify and/or add data to the Oracle database through container instances of the AMI. Modifications to extensional monitored containers will awaken Oracle triggers continuing the process.

4.4 Control Flow

Figure 4 also illustrates the control flow of VenusIDS. As previously stated, data enter the system through a bank of monitored networks causing the Oracle database to be updated. Oracle then evaluates which tables have been updated, and passes control to the applicable trigger code[5]. The trigger code then spawns a C program that sends data to an AF_UNIX socket. When the trigger code finishes, the transaction on the database side is completed.

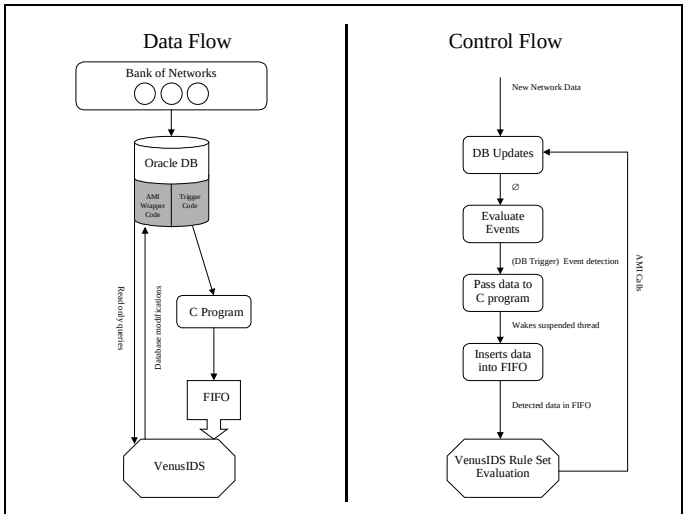

Figure 4. VenusIDS Layered Architecture

The VenusIDS component contains a detached thread that monitors an AF_UNIX socket for data and inserts detected data into a shared, semaphore-protected FIFO data-structure. Data are removed from the shared FIFO queue and used to call Venus modules. AMI calls then may modify the database and start the process over again. Note that since the database and the rule-set are in different address spaces, rule execution occurs in parallel with the database. This tightly layered approach follows the same semantics as a batch oriented Venus programs.

4.5 Code Modules

VenusIDS is composed of five code units:

- A refined AMI container definition - This contains the code that communicates with the Oracle database and additionally implements a detached thread that inserts data into an associated FIFO buffer.
- Oracle trigger code - This code passes data to a C program using a non-standard code escape.
- A C program that passes data through a socket call interface - This component must be written in a general-purpose programming language. It cannot be written in SQL trigger code because of its communication via system sockets.
- A semaphore-protected FIFO data structure.
- The VenusDB rule code implementing the analysts' knowledge in intrusion detection.

Now that we have a hand-generated implementation in hand, future releases of the VenusDB compiler will be extended to output the first four code units described above.

[5] We would like for this evaluation to be done upon database commit. We believe this should be a standard option of trigger semantics. Otherwise, the rule-system may be notified of an insertion that is later aborted and evaluate rules on incomplete relations.

Time (seconds)	Empty DB	10,000 rows	100,000 rows	100,000 optimized
Average	.004	.04	.8	.3
Minimum	.0002	.0002	.0002	.0002
Maximum	.8	6.	56.	26.
Std. Deviation	.04	.41	4.7	2.3

Table 1. VenusIDS Performance Metrics

5 Performance

We measure two system properties: latency and transaction time[6]. The latency in the system occurs between the insertion of a row into a table until the completion of its associated trigger action. Using VenusIDS, this latency was measured at 20 milliseconds with the measurement being constant regardless of the database size. The second aspect of latency that was measured is the latency of inserting a row, writing the row to the socket, retrieving it from the socket and placing it into the FIFO, and removing it from the FIFO. This latency was measured at 23 milliseconds. Thus, the entire event detection scheme nominally adds 3 milliseconds. Note, however, that there are several distinct operating system processes involved in this scheme. Thus, there may be some amount of distributed execution, and similar results may not be achievable on a uniprocessor.

The second measurement made was the transaction time; the time to execution (per event) for the VenusDB portion of VenusIDS. This time necessarily varies because the amount of work done by the rule system depends upon the modified row and the state of the database. Evaluation may include chained rules and each rule may require multiple database queries executed through the AMI. To examine transaction time, we initiated a data flow of 1,000 connections and 1,000 suspicious packet records. A subset of these records contained values sufficient to cause rule firings. The initial state of the database at the beginning of the insertion flow is a test parameter. Results are shown in Table 1. In the first experiment, the database was empty. In the second, it contained 10,000 preexisting rows. In the third, it contained 100,000 rows. As shown by the uniformly low minimum transaction time, database inserts that do not trigger rule firings are quickly evaluated and discarded. The average and maximum transaction times roughly scale linearly with database size.

A measure of VenusIDS's usefulness is the speed at which alarms are logged. The goal is to process alarms at or above the frequency they are logged from the component networks. Network analysts have examined a maximum rate of intrusions from component sensors, the events in which VenusIDS monitors, near 60 alarms per hour or 0.02 events per second. Our results demonstrate that the most heavily loaded database we measured should be able to process more than one event per second, about 60 times the event rate of our current system. Further, the extensional monitored containers within VenusIDS average less than 100,000 rows and few rules call for general queries on the large connection log.[7] Thus, our preliminarily results demonstrate that VenusIDS does succeed in processing events fast enough to satisfy the requirements of a near real time intrusion detection system.

6 Future work

This section describes the areas we are investigating for future work. Areas include performance improvements and data sources research.

[6] The performance tests were performed on a Sun Ultra-2 dual CPU machine with 256 megabytes of main memory and a 6-disk RAID Level 5 storage unit. Its SPECint95 rating is 12.3 [9].

[7] Only the connection log is extremely large, and connections are logged in batch mode, offline, as described in Section 2.

6.1 Performance Improvements

Research into performance improvements can be divided into two sub-components: 1) optimizing the VenusDB compiler; and 2) the investigation of parallel rule sets.

6.1.1 VenusDB Optimizations

The extension of Venus to active database execution has provided many optimization opportunities. Among the opportunities are the optimization techniques described in [15] to optimize both the target code per the physical schema of the application database and to suggest how to augment the physical schema to enable further performance enhancements. Repeated cycles of this optimization are expected to yield performance improvements as the VenusDB architecture adaptively specializes its target code with respect to these schema improvements while exploiting VenusDB's predicate pushdown facility. The dynamically executed predicates that are pushed to the component databases can themselves be optimized.

Evidence of the benefits of such optimizations is found in Table 1. After a careful examination of the 100,000 row behavior, we hand-tuned the database's index structures and the AMI implementation. These optimizations reduced execution times by approximately half.

6.1.2 Parallel Rule Sets

Almost all modern commercial databases have support for multiple users. We would like to take advantage of this capability by executing rule sets in parallel. This can be accomplish by prioritizing events and their associated rule-sets and providing each rule set with its own dedicated FIFO. Oracle triggers will then insert data into the FIFO based on table type. VenusIDS will then read from the FIFO's in priority order.

This prioritization will allow us to better exploit parallel models. Events that spawn expensive transactions could be executed in a different process from known efficient event transactions. This would have the dual effects of increasing system bandwidth and reducing the average length of the FIFO's and waiting time for known inexpensive events. Additionally, if mean time transaction processing and database update rates can be determined, a queuing model can be established and the execution can be scheduled by an asynchronous real-time operating system with hard real-time constraints as described in [2,3,7].

6.2 Data Sources Research

Research into the data sources that supply our analysis layer can be divided into two sub-components: 1) investigation of distributed data sources; and 2) data formatting issues.

6.2.1 Distributed Data Sources

VenusIDS's implementation using VenusDB provides a robust method to integrate distributed data sources directly within our analysis layer through AMI instances. Further, no modification of the underlying data sources will be required. We plan to utilize this ability to take advantage of the many intrusion detection systems that are currently available.

Our experience is that there are many common storage units in which intrusion detection systems log network traffic. Though we encourage the use of commercial databases, we embrace this dissimilarity rather than discourage it. The use of multiple storage units 1) distributes the data making the data less vulnerable; 2) reduces the resources needed by the analysis layers; and 3) is naturally exploited for parallelism and optimization via VenusDB's predicate pushdown facility. The challenge here is to discover how much we can take advantage of distributed sources and to decide what resources and architectures are the most useful. Towards this end, an investigation into hierarchical intrusion detection systems is warranted.

6.2.2 Data Formatting Issues

One of the major complexities of any expert system is knowledge acquisition and the implementation of that knowledge. A severe limitation has surfaced. The data formats encompassing

network traffic have been found to be mostly insufficient for the purpose of network intrusion detection rules. Most intrusion detection systems log network traffic containing discrete measures (such as to and from IP addresses, the time of the connection, login destination, etc) and continuous measures (such as CPU time, IO activity, etc) [5]. Unfortunately, encoding of the data transferred during a connection and the events recognized by probes are not nearly as informative. This information is often contained solely within ASCII. Thus, many of our rules must, at run-time, parse unstructured ASCII data to discover anomaly patterns. Consequently, this lack of structure severely limits the quality and extensibility of VenusIDS rules.

Our knowledge engineers are working on more effective strategies to structure event and connection information data. Two options have surfaced; 1) structuring the ASCII streams and 2) defining a set of enumerated tags describing the streams and events. At this juncture, neither option has proven to excel over the other.

7 Conclusion

This paper presents VenusIDS, a layered active database implementation of the analysis layer of a two-layered network intrusion detection system. The use of active database technology and the highly optimized VenusDB compiler together yield near real time performance. Further, our preliminary performance results demonstrate that VenusIDS is scalable with the event rate of our two-layered IDS.

VenusIDS, due to its implementation using VenusDB, is itself a layered architecture that provides for heterogeneity. Thus, rules within VenusIDS refer to distributed (database) data sources without any special rule syntax or modifications to underlying components. Together, the combination of VenusIDS's declarative expression of rules, tight integration with databases and efficient performance yields a powerful near real time intrusion detection tool which readily supports complex intrusion detection rules generated by domain experts and/or automatic methods. Future performance improvements and advances in parallel computation are expected to support a truly schedulable real time-intrusion detection system.

8 Bibliography

1. J.C. Browne, et. al. A New Approach to Modularity in Rule-Based Programming. In *Proceedings of the 6th International Conference on Tools with Artificial Intelligence*, IEEE Press, 1994, 18-25.
2. S. Correl, D.P. Miranker. "On Isolation, Concurrency, and the Venus Rule Language." In *Proceedings of the 4th International Conference on Information and Knowledge Management*, November 1995.
3. E. Doğdu, G. Özsoyoğlu. Adaptive Real-Time Transactions and Risk-based Load Control. In the *Workshop on Databases: Active And Real-time 1996*, DART '96. Rockville, Maryland, November 15, 1996.
4. C.L. Forgy. *OPS5 User's Manual*. Technical Report CMU-CS-81-135, Department of Computer Sciences, Carnegie Mellon University, July 1981.
5. Teresa F. Lunt, R. Jagannathan. A Prototype Real-Time Intrusion-Detection Expert System. *IEEE Computer Society Symposium on Research in Security and Privacy*, 1988.
6. D.P. Miranker, D.Brant, B.J. Lofaso Jr., and D. Gadbois. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, AAAI, July 1990, 685-692.
7. A. Mok, F. Wang, and E. A. Emerson. Distributed real-time system specification and verification in APTL. *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 4, October 1993, pp. 346-378.
8. L. Obermeyer. *Abstractions and Algorithms for Active Multidatabases*. Ph.D. Thesis, The University of Texas at Austin, March 1999.
9. Standard Performance Evaluation Corporation, 1998.
10. W. Lee, S.J. Stolfo, and K. Mok. A Data Mining Framework for Building Intrusion Detection Models. To appear in the *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.
11. Ullman, Jeffrey D. *Principles of Database & Knowledge-Base Systems Vol. 1*. W H Freeman & Co.; ISBN: 0716781581, December 1988.
12. J. Widom, S. Ceri. *Active database Systems, Triggers and Rules For Advanced database Processing*. Morgan Kaufmann Publishers, Inc. San Francisco, CA, 1996.

13. G. Vigna, R. A. Kemmerer. NetSTAT: A Network based Intrusion Detection Approach. In Proceedings of the 1998 ACSOC.
14. Lane B. Warshaw, Daniel P. Miranker. A Case Study of Venus and a Declarative Bases for Rule Modules. In *Proceedings of the 5th Conference on Information and Knowledge Management*, November 1996.
15. Lane B. Warshaw, Dan Miranker. Rule-Based Query Optimization, Revisited. Submitted to the Conference on Information and Knowledge Management, 1999.