Rapid Trust Establishment for Pervasive Personal Computing

Trust-Sniffer's staged approach to establishing confidence in untrusted machines balances security and ease-of-use, facilitating rapid use of transient hardware.

Ajay Surie, Adrian Perrig, Mahadev Satyanarayanan, and David J. Farber
Carnegie Mellon University

In the emerging Internet Suspend/Resume mobile computing model,[1–3] users exploit pervasive deployments of inexpensive, mass-market PC hardware rather than carrying portable hardware. ISR's driving vision is that plummeting hardware costs will someday eliminate the need to carry computing environments in portable computers. Instead, ISR will deliver an exact replica of the last checkpointed state of a user's entire computing environment (including the operating system, applications, files, and customizations), on demand over the Internet to hardware located nearby. So, ISR cuts the tight binding between PC state and PC hardware. ISR is implemented by layering a virtual machine on distributed storage. The virtual machine encapsulates execution and user customization; the distributed storage transports that state across space and time.

In this new computing model, establishing trust in unmanaged hardware for transient use becomes a major issue. Today, when a user sits down at a computer in the office or home, he or she implicitly assumes that the machine hasn't been tampered with and that it doesn't contain malware, such as a keystroke logger. This assumption is reasonable because unauthorized physical access to the machine is restricted. The same assumption applies to a portable computer that the user physically safeguards at all times. For ISR's vision of transient hardware use to become commonplace, users must be able to quickly establish a similar confidence level in hardware that they don't own or manage.

To address this problem, we've created *Trust-Sniffer*, a tool that helps users incrementally gain confidence in an initially untrusted machine. Trust-Sniffer focuses on software attacks but doesn't guard against hardware attacks, such as modifying the basic I/O system. Potential defenses against hardware attacks include physical surveillance or the use of tamper-proof or tamper-evident hardware. Although Trust-Sniffer is designed to safeguard mobile users, you can couple it with system operators' security technologies, such as full disk encryption, direct memory-access monitoring, and remote attestation.

Trust-Sniffer's staged approach to establishing confidence in an untrusted machine balances the needs for security, usability, and speed (for a comparison of Trust-Sniffer and other approaches, see the sidebar).

Design overview

Trust-Sniffer aims to enhance security with modest user effort. A key design principle, motivated by ISR's unique characteristics, is to validate only the software a user needs for a task. Specifically, most of a user's execution environment is fetched from a trusted server over an authenticated, encrypted channel. This includes the guest operating system and applications that execute inside the user's virtual machine. Cached

Published by the IEEE Computer Society ■ 1536-1268/07/$25.00 © 2007 IEEE

IPR2023-01464
CrowdStrike EX1011 Page 1

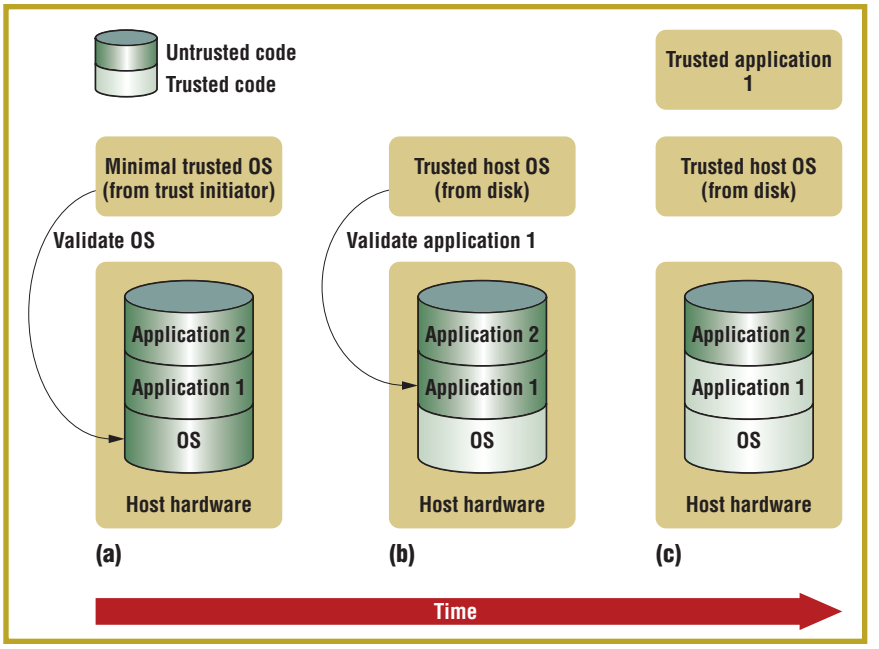


Figure 1. Trust-Sniffer's staged approach to trust establishment. Initially, all software is untrusted. (a) The trust initiator's boot uses a minimal trusted operating system to validate the on-disk OS. (b) Next, the trusted host OS is booted from disk, which validates applications as required. (c) The host OS only permits trusted applications to execute.

virtual-machine state on disk is always encrypted, and ISR verifies its integrity before a virtual machine uses it. So, it's only necessary to verify the integrity of a small core of local ISR and Linux software. Other compromised state isn't a threat if it's never used when a machine is functioning as an ISR client. This minimalist approach speeds trust establishment and makes ISR more ubiquitous. A second important design principle is to prevent the execution of untrusted software. This is in contrast to attestation techniques, which facilitate the detection of untrusted software but don't prevent its execution.

Staged approach

Figure 1 shows Trust-Sniffer's staged approach to trust establishment.

Establishing a root of trust. First, the user performs a minimal boot of the untrusted machine from a small, lightweight device such as a USB memory stick that the user owns and carries at all times. This *trust initiator* device serves as the root of trust in Trust-Sniffer. Relying on a trusted physical possession is convenient and easy for the user to understand. This stage involves a minimal boot, because its sole purpose is for Trust-Sniffer to examine the local disk and verify that it's safe to perform a normal boot using the on-disk operating system and its associated boot software. The minimal boot ignores the network and any devices other than the disk.

Booting the on-disk operating system. Next, Trust-Sniffer performs a normal reboot from disk, which is now known to be safe. This ensures that a full suite of drivers for the local hardware (such as graphics accelerators) as well as correct local environment settings (such as those for printers, networks, and time) is obtained. During this boot process, Trust-Sniffer dynamically loads the *trust extender* module into the kernel, which extends the zone of trust as execution proceeds. On the first attempt to execute any code that lies outside the current zone of trust (including dynamically linked libraries), the kernel triggers a *trust fault*. The trust extender handles a trust fault by verifying the suspect module's integrity and stops execution if the module's integrity cannot be established.

Validating other local software on trust faults. As each component of ISR client software is accessed for the first time, the kernel generates trust faults and Trust-Sniffer validates the component's integrity. Once the user gains confidence in the machine and its ISR software, he or she can perform the resume step of ISR. On the other hand, if Trust-Sniffer can't validate any component of ISR client software, it alerts the user and terminates the ISR resume sequence.

On-demand validation is much more robust than en-masse validation with respect to ISR software evolution. If a new release of ISR client software uses a local software component that previous releases didn't use, Trust-Sniffer will discover its use, even if a software developer fails to list it.

Example use

The following example illustrates our system's use. Bob boots up a PC at his hotel's business center using his USB key. Trust-Sniffer's initial scan quickly validates the on-disk operating system, which is subsequently booted. Bob then initiates the resume step of ISR. Once Trust-Sniffer verifies the ISR client software's integrity, ISR fetches Bob's personal execution environment over a trusted communication channel. The locally installed Web browser is riddled with malware, but this doesn't affect Bob's task because he works only within his trusted personal execution environment.

Threat model and assumptions

Machines used as ISR clients are vulnerable to attacks such as modifications to client or system software and installation of malware such as key-logging or screen-capture software. Trust-Sniffer's goal is to avoid potential loss or disclosure

IPR2023-01464
CrowdStrike EX1011 Page 2

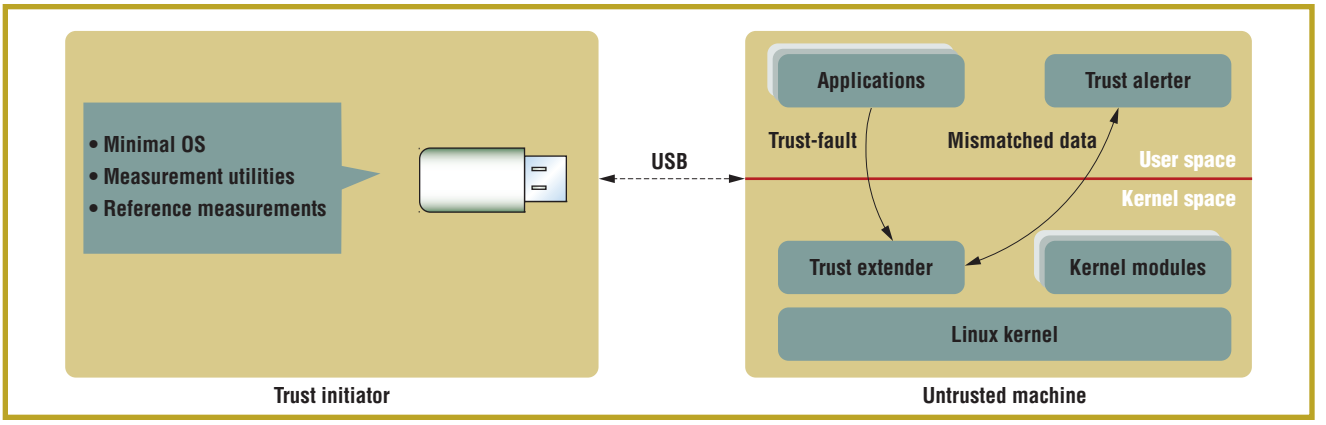


Figure 2. The Trust-Sniffer architecture. The trust initiator plugs into an untrusted machine, validates its operating system, and equips it with a list of trusted applications' SHA-1 hashes (reference measurements). Applications that haven't been validated cause a trust fault, which the trust extender handles. If an application can't be validated, the user-space trust alerter notifies the user.

of user data by validating software on an ISR client machine before a user accesses his or her personal execution environment.

We assume that, to use our system, the user may reboot the untrusted machine and that the BIOS allows booting from a USB memory stick. We also assume that modifying the BIOS is difficult. So, we don't guard against virtual-machine-based attacks, such as SubVirt,[4] where a compromised BIOS could boot directly into a malicious virtual-machine monitor.

In addition, Trust-Sniffer is based on *load-time* binary validation, so it doesn't protect against runtime attacks. Specifically, only applications with valid signatures may load and execute, but Trust-Sniffer wouldn't detect, for example, buffer overflow attacks initiated after execution has begun.

Detailed design and implementation

Figure 2 shows Trust-Sniffer's three major components:

- the trust initiator and its associated minimal boot software;
- the trust extender, implemented as a kernel module; and
- the trust alerter, a user-space notifier application.

Integrity measurement architecture

Trust-Sniffer builds on the implementation of Integrity Measurement Architecture for Linux.[5] IMA checks the system software stack's integrity by computing a SHA-1 hash over an executable's contents when it's loaded. This SHA-1 hash is called a *measurement*. IMA is built into the kernel and is invoked whenever executable code is loaded, accounting for user-level binaries, dynamically loadable libraries, kernel modules, and scripts. IMA measures each loaded executable and stores its measurement in a list in the kernel. An aggregate measurement of the list is stored in the Trusted Platform Module, a secure hardware coprocessor that protects the list's integrity. The TPM aggregate and the kernel-measurement list facilitate remote-party attestation of the system software stack.

IMA is part of the kernel and assumes trust in all software that's executed before it's invoked. Our system builds on IMA and addresses these concerns. Because remote-party attestation isn't our system's intent, we don't use a TPM chip. Instead, we use a small, user-carried passive device to initiate the establishment of confidence in the boot loader and kernel on the untrusted machine. The machine's OS kernel is then responsible for measuring software applications and preventing untrusted software from executing.

Validating applications

Trust-Sniffer validates an application by comparing its *sample* measurement, obtained when the application is loaded, to a known list of *reference* measurements generated from trusted applications. A *mismatch* is an inequality between a sample measurement and every reference measurement in the list. In contrast to attestation, where measurements detect untrusted software after it has been loaded, *validation* refers to detection of untrusted software before it's used.

The reference measurement list must be generated initially and updated only when patches and new software are released. Failing to update the measurement list implies that new trusted software will be prevented from executing. Execution of untrusted software is never allowed. For simplicity and comparability, our measurement list format is the same as that output by `sha1sum`, a utility commonly available on most distributions.

In our initial implementation, we recorded measurements for the latest version of an application. However, the list could contain multiple trusted measurements for an application such as the kernel. In this case, if Trust-Sniffer encountered a system with multiple kernels, it could permit the execution of a trusted kernel even if the remaining ones

IPR2023-01464
CrowdStrike EX1011 Page 3

Related Work in Trust Establishment

When comparing Trust-Sniffer to other work, we considered the following:

- Most systems are designed with an administrator in mind and don't focus on giving users security guarantees.
- Although some solutions offer more security than our system, their complex hardware and software requirements limit their adoption in practice.

The *trusted boot* and *secure boot*[1] mechanisms verify software components required during a PC's bootstrap process. However, both mechanisms require platform modifications. Neither mechanism verifies software, such as operating system services and user applications, which are executed after the bootstrap process. Additionally, although trusted boot helps detect untrusted software through recorded application signatures, it doesn't prevent untrusted execution because it doesn't validate the signatures it records.

Trust-Sniffer differs from solutions such as SoulPad, which lets users configure a machine with their own personal computing environment using software carried on a portable device.[2] This approach isn't always practical because personalized environments often don't contain correct local-environment settings, such as printer and network settings, or appropriate drivers for the target machine. Trust-Sniffer requires minimal boot software and subsequently loads software on the target system after establishing its integrity.

A well-studied problem is the *untrusted terminal*, where a user interacts with an untrusted device. The most interesting formulation of the problem is using an untrusted device to establish secure communication with a trusted remote device. Researchers have proposed several solutions, including camera-based authentication,[3] visual cryptography,[4] and trusted smart cards.[5] Other techniques to verify the authenticity of a system's configuration include a challenge-response mechanism between a trusted authority and a remote system.[6] The Pioneer system uses a similar protocol to provide verifiable code execution on an untrusted platform.[7] Because Pioneer establishes a dynamic root of trust, it can function as a building block for future iterations of Trust-Sniffer.

Scott Garris and his colleagues describe a mechanism for a user to establish trust in an untrusted kiosk—a problem similar to the one Trust-Sniffer addresses.[8] However, their implementation relies on secure hardware, including a Trusted Platform Module chip, and a new instruction recently added to the x86 architecture. In addition, their implementation requires validating all the software on a kiosk before it can be used, whereas Trust-Sniffer allows incremental program validation to facilitate transient use.

REFERENCES

1. W.A. Arbaugh, D.J. Farber, and J.M. Smith, "A Secure and Reliable Bootstrap Architecture," *Proc. 1997 IEEE Symp. Security and Privacy*, IEEE CS Press, 1997, pp. 65–71.
2. R. Cáceres et al., "Reincarnating PCs with Portable SoulPads," *Proc. 3rd Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 05), ACM Press, 2005, pp. 65–78.
3. D.E. Clarke et al., "The Untrusted Computer Problem and Camera-Based Authentication," *Proc. 1st Int'l Conf. Pervasive Computing*, LNCS 2414, Springer, 2002, pp. 114–124.
4. M. Naor and B. Pinkas, "Visual Authentication and Identification," *Proc. 17th Ann. Int'l Cryptology Conf. Advances in Cryptology*, LNCS 1294, Springer, 1997, pp. 322–336.
5. M. Abadi et al., "Authentication and Delegation with Smart-Cards," *Science of Computer Programming*, vol. 21, no. 2, 1993, pp. 91–113.
6. R. Kennell and L.H. Jamieson, "Establishing the Genuinity of Remote Computer Systems," *Proc. 12th Conf. Usenix Security Symp.*, Usenix Assoc., 2003, pp. 295–310.
7. A. Seshadri et al., "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems," *Proc. 20th ACM Symp. Operating Systems Principles*, ACM Press, 2005, pp. 1–16.
8. S. Garris et al., "Towards Trustworthy Kiosk Computing," *Proc. 8th IEEE Workshop Mobile Computing Systems and Applications*, IEEE CS Press, 2007.

were compromised. Because ISR client software fetches most of a user's execution environment, Trust-Sniffer must maintain measurements for only a small set of OS and client software. To address frequent software updates and patches, users could obtain new measurements from a server that periodically generated updated lists and digitally signed them for distribution.

Rapidly establishing a root of trust

As we mentioned earlier, none of the software on the target machine is initially trusted. The trust initiator aims to

- bootstrap the trust-establishment process by establishing a root of trust and
- equip the on-disk OS with the necessary tools to validate the rest of the software on the machine.

It does this as rapidly as possible to avoid a long delay before the user can do useful work.

The trust initiator is partitioned, formatted, and loaded with the Finnix bootable OS, a derivative of Knoppix. Finnix provides both excellent support for devices and automatic hardware detection. It's suitable for our purposes because it boots quickly and has a small

IPR2023-01464
CrowdStrike EX1011 Page 4

footprint. The trust initiator is loaded with the list of reference measurements for the initial boot-validation phase as well as subsequent application validation by the on-disk kernel.

After the machine is booted using the trust initiator, all on-disk software components associated with the boot process are validated. A custom start-up script mounts local hard disks and discovers the boot partition. It then uses sha1sum to measure the kernel image, the initial ram disk, and the GRUB boot loader. Trust-Sniffer compares the measurements of this set of software to the trust initiator's list of reference measurements. Any mismatches cause the boot process to be untrusted and to halt trust establishment because a trusted OS is required to validate the rest of the software stack. If all the boot software is valid, Trust-Sniffer copies the trust initiator's reference measurement list to a predetermined location on disk, which the kernel subsequently accesses when it boots.

Dynamically extending the root of trust

Next, the trust extender validates application software. We avoid making the user manually reboot (that is, powering off the machine, removing the trust initiator, and then powering the machine on) by using kexec, which lets you boot into a new kernel directly from the current running kernel without using a boot loader.

The trust extender kernel module uses the securityfs pseudo file system to communicate with user-space programs. This provides an interface for a user-space program to load the trust extender with the trust initiator's list of reference measurements.

The trust extender can begin enforcing measurement mismatches only after the reference measurement list has been loaded. So, it's imperative to load this list as early in the boot process as possible. Until the list is loaded, the trust extender is nonintrusive and allows all execution. We use a custom initrd (initial RAM disk) image with modified boot scripts to load the reference measurements into the trust extender. All initial execution before the list is loaded is from the initrd, which is part of the root of trust. Once the reference measurements are loaded, the trust extender protects the in-kernel list's integrity by disabling the measurement-loading interface in securityfs.

The reference measurement list resides in a hash table indexed by each executable's SHA-1 hash value. The trust extender uses the file_mmap Linux Security Module hook to measure files mapped with the PROT_EXEC bit set (which includes dynamically loadable libraries and executables) and a custom hook in the load_module function to measure kernel modules. The LSM interface is part of the main kernel. Each measurement consists of the executable code's SHA-1 hash value. The trust extender also stores additional file metadata such as pathname, user ID, and group ID. The metadata doesn't affect the actual measurement validation, but we use it to communicate information to the user (see the next section for more details). When a trust fault triggers the measurement of an executable, its value is compared against the reference list. If no matching reference measurement exists, the measurement is untrusted, and the kernel disallows the execution.

The system's caching mechanism reduces the overhead of measuring unchanged applications when they're executed more than once. When a previously measured application is executed again, the kernel simply uses the application's cached measurement value, unless it's stale (that is, unless the file is opened for writing or is on an unmounted file system). Stale measurements must be recomputed.

The system's caching mechanism reduces the overhead of measuring unchanged applications when they're executed more than once.

Alerting the user

When an untrusted application is encountered, the kernel blocks its execution. The user isn't aware of communication between applications and the kernel and must be alerted to relevant information in two instances: when the trust extender encounters an untrusted application, and when a failure occurs in the kernel that would compromise the measurement process. The latter case encompasses unexpected failures in the measurement process, such as out-of-memory errors.

To establish a communication channel between the trust alerter and the trust extender, we used the netlink socket interface. The netlink interface is a bidirectional, versatile method for passing data between the kernel and user space, and the interface is well documented. We installed a new netlink protocol type, TS_NETLINK, and added the TS_NLMSG_MISMATCH and TS_NLMSG_FAILURE message types. When the trust extender is initialized, it opens a netlink socket using a call to netlink_kernel_create. A user process that binds to this socket waits for messages from the kernel using the recvmesg system call.

When the trust extender encounters an untrusted application, it sends the appropriate message with a call to netlink_

IPR2023-01464
CrowdStrike EX1011 Page 5

broadcast. In the case of a mismatch, this includes the process ID of the process that failed and the file name of the application on disk, if available. We've implemented a simple console application that communicates with the kernel in the current implementation. We plan to convert this into a friendlier GUI application in the future.

Evaluation

We evaluated Trust-Sniffer in light of user expectations about the system's security guarantees and performance.

Security

We needed an appropriate balance between ease of use and security guarantees. Currently, users have no choice but to trust any unmanaged hardware that they encounter. Trust-Sniffer provides a simple solution designed for use by nonexperts and security guarantees that are a substantial improvement over current practice.

Trust-Sniffer isn't foolproof. It's vulnerable to certain attacks. As we mentioned earlier, we don't guard against a malicious BIOS or virtual-machine attack. We also assume that the hardware is trustworthy, so we don't protect against direct-memory access (DMA) based attacks by malicious devices.

Although Trust-Sniffer disallows the execution of unknown software, this shouldn't inconvenience users. Establishing trust in a machine for use as an ISR client requires validation of only a small subset of software because most applications are contained in the user's personalized execution environment that's downloaded from a trusted server. It's easy to equip the trust initiator with reference measurements required to validate an ISR client's software.

Because the trust initiator is write protected, unauthorized external sources can't modify its software. Of course, users can disable write protection on their own machines to update the reference measurement list. In addition, the trust establishment procedure doesn't depend on network communication with the untrusted machine. So, we disable networking capabilities on the minimal OS that we use to validate the on-disk kernel.

Performance

Our prototype implementation used the Fedora Core 5 distribution, configured with version 2.6.15 of the Linux kernel. We implemented Trust-Sniffer by extending a core part of the Linux kernel, so configuration changes (such as using a different Linux distribution) shouldn't significantly affect the results. Our test hardware consisted of an IBM T43 laptop with a 2.0 GHz Pentium M processor and 1 GB of RAM. We used a standard USB memory stick with 1 GB of storage capacity as the trust initiator.

From the transient usage model's perspective, we evaluate how much overhead the Trust-Sniffer system requires compared to a system without security checks. The user cares most about how long the system takes to boot and the overhead it needs to check applications. With Trust-Sniffer, the downtime caused by having to reboot is small and the additional security is worth the performance penalty. Note that the performance overhead for application measurements is only on the first execution.

Because the trust extender primarily uses the file_mmap LSM hook to measure applications at load time, we measure an mmap operation's latency in different contexts using the HBench-OS framework.[6] We averaged latency measurements over 10 runs of the benchmark.

When an application is executed for the first time, the system computes the application's SHA-1 measurement, which we denote as a *cache miss*. If on subsequent executions an application's cached measurement is valid, its execution results in a *cache hit*. In our experiments using a system without Trust-Sniffer, an mmap operation's latency was 0.99 μs. A cache hit had a latency of 1.19 μs, with an overhead against the baseline of 0.20 μs. A cache miss had a latency of 4.29 μs, with an overhead of 3.3 μs. Clearly, with a cache hit, the overhead of validating a previously measured application isn't significant. When an application is first executed, there's some measurement overhead compared to the baseline. However, this overhead is low in absolute terms because applications are measured only once.

To evaluate the boot process, we measured the average time (over five trials) it took to boot a machine from when we pushed the power button until a logon prompt appeared. This metric is relevant to the user because it defines the "warm-up" time the system requires before the user can start working. These results depend somewhat on system configuration, such as the number of daemons started when the OS is booted. However, they illustrate sufficiently that the overhead of using Trust-Sniffer is small. A standard machine without Trust-Sniffer takes 97.1 seconds to boot; a machine with Trust-Sniffer takes 111.4 seconds. The overhead of using Trust-Sniffer is

We needed an appropriate balance between ease of use and security guarantees. Trust-Sniffer provides security guarantees that are a substantial improvement over current practice.

IPR2023-01464
CrowdStrike EX1011 Page 6

thus only 14.2 percent over a standard system. One might argue that users don't usually have to reboot a system. However, we believe that the additional time spent to establish trust is well worth the security improvement.

Usability and extensibility

In computing systems, very often a solution's complexity hinders its acceptance in practice. We focused our design of Trust-Sniffer on simplicity and ease-of-use to facilitate adoption by nonexpert users. With Trust-Sniffer, the zone of trust expands from a small, convenient, trusted device that the user carries. Trust-Sniffer's operation model can help increase awareness and security for simple day-to-day computing tasks.

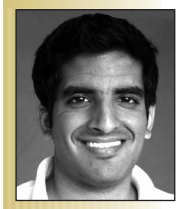
Our design is flexible and extensible. It should be easy to set up a mechanism for users to obtain updates to reference measurement lists. Our prototype used Fedora Core; however, because we implemented the trust extender as a kernel module, the trust initiator could be loaded with appropriately configured kernel modules and initial ram disks for stock kernels of various distributions. This would make it possible to equip arbitrary machines that don't have preconfigured kernels with Trust-Sniffer software.

Trust-Sniffer provides greater overall security for users and increases their awareness about security risks. In future versions, we plan to explore ways to provide increased security while keeping the design focused on nonexpert users. ■

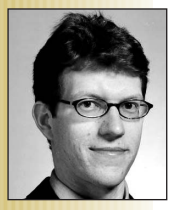
ACKNOWLEDGMENTS

We thank Reiner Sailer of IBM Research for his assistance with the Integrity Measurement Architecture and Jan Harkes for his suggestions on the project. This research was supported by the US National Science Foundation under grant CNS-0509004, the US Army Research Office under grant DAAD19-02-1-0389 to Carnegie Mellon's CyLab, and the Intel Corporation.

the AUTHORS

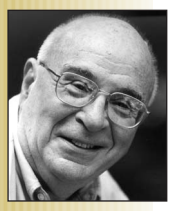


Ajay Surie is a graduate student in computer science at Carnegie Mellon University. His research interests include distributed systems and security. He received his BS in computer science from Carnegie Mellon University. Contact him at the Computer Science Dept., Carnegie Mellon Univ., Wean Hall 4212, 5000 Forbes Ave., Pittsburgh, PA 15213; asurie@cs.cmu.edu.



Adrian Perrig is an assistant professor in electrical and computer engineering at Carnegie Mellon University. His research interests include networking and systems security and security for mobile computing and sensor networks. He received his PhD in computer science from Carnegie Mellon University. He's a member of the ACM and IEEE and a recipient of the NSF Career award, an IBM faculty fellowship, and a Sloan research fellowship. Contact him at 2110 Collaborative Innovation Center, 4720 Forbes Ave., Pittsburgh, PA 15213; adrian@ece.cmu.edu.

Mahadev Satyanarayanan is the Carnegie Group Professor of Computer Science at Carnegie Mellon University. His research interests include mobile computing, pervasive computing, and distributed systems. He received his PhD in computer science from Carnegie Mellon University. He's a fellow of the ACM and IEEE and the founding editor in chief of *IEEE Pervasive Computing*. Contact him at the Computer Science Dept., Carnegie Mellon Univ., Wean Hall 4212, 5000 Forbes Ave., Pittsburgh, PA 15213; satya@cs.cmu.edu.



David J. Farber is a Distinguished Career Professor of Computer Science and Public Policy at Carnegie Mellon University. His research interests include networking, distributed systems, and public policy. He received an honorary doctor of engineering degree from the Stevens Institute of Technology. He's a fellow of the ACM and IEEE. Contact him at the Inst. for Software Research Int'l, Carnegie Mellon Univ., 5000 Forbes Ave., Pittsburgh, PA 15213; dfarber@cs.cmu.edu.

REFERENCES

1. M. Kozuch and M. Satyanarayanan, "Internet Suspend/Resume," *Proc. 4th IEEE Workshop Mobile Computing Systems and Applications*, IEEE CS Press, 2002, p. 40.
2. M. Satyanaranyanan et al., "Towards Seamless Mobility on Pervasive Hardware," *Pervasive and Mobile Computing*, vol. 1, no. 2, 2005, pp. 157–189.
3. M. Satyanaranyanan et al., "Pervasive Personal Computing in an Internet Suspend/Resume System," *IEEE Internet Computing*, vol. 11, no. 2, 2007, pp. 16–25.
4. S.T. King et al., "SubVirt: Implementing Malware with Virtual Machines," *Proc. 2006 IEEE Symp. Security and Privacy*, IEEE CS Press, 2006, pp. 314–327.
5. R. Sailer et al., "Design and Implementation of a TCG-Based Integrity Measurement Architecture," *Proc. 13th Conf. Usenix Security Symp.*, Usenix Assoc., 2004, pp. 223–238.
6. A.B. Brown and M.I. Seltzer, "Operating System Benchmarking in the Wake of LMbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," *Proc. 1997 ACM Sigmetrics Int'l Conf. Measurement and Modeling of Computer Systems*, ACM Press, 1997, pp. 214–224.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.

IPR2023-01464
CrowdStrike EX1011 Page 7