# Rapid Trust Establishment for Transient Use of Unmanaged Hardware

**Ajay Surie, Adrian Perrig, M. Satyanarayanan, David Farber**

December 2006
CMU-CS-06-176

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

Transient use of PCs has grown in importance with the advent of Internet cafes and the emergence of personalization systems such as Migo, GoToMyPC, and Internet Suspend/Resume.® Unfortunately, users have no choice today but to trust any transient hardware they use. They are often unaware of the risks they face in placing faith in public computers. We address this problem through Trust-Sniffer, a tool that helps a user to gain confidence in the software stack on an untrusted machine. The root of trust is a small, lightweight device such as a USB memory stick that is owned by the user. Once the integrity of the boot image is verified, Trust-Sniffer uses a staged process to expand the zone of trust. It generates a *trust fault* when a user first attempts to execute any binary that lies outside the current zone of trust. A trust fault handler verifies the integrity of the suspect binary by comparing its checksum with that of known good binaries. Execution stops if the binary's integrity cannot be established. This staged approach to establishing confidence in an untrusted machine strikes a good balance between the needs of security and ease-of-use, and enables rapid use of transient hardware.

This research was supported by the National Science Foundation (NSF) under grant number CNS-0509004, and by the Army Research Office (ARO) through grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, ARO or Carnegie Mellon University.

**Keywords:** Establishing trust, untrusted terminal, transient use, trusted computing, load-time validation, security, integrity, portable storage

# 1 Introduction

Establishing trust in computing platforms is becoming more important as users begin to access machines in multiple locations. Even for common day-to-day user tasks, such as web browsing, online shopping, checking email, or editing documents, the risk of a user's personal data being compromised by a malicious system is high. For example, at home, a shared computer could become compromised through a child's unwitting download of a virus over the Internet. Public computers such as those in a cluster, an Internet cafe or a hotel could easily be rendered unsafe by an attacker installing malicious software, such as a keystroke logger. Many users do not realize the security risks that they face on an everyday basis, let alone take adequate measures to protect against such threats.

There are a wide range of technologies available today that aim to provide ubiquitous access to a user's computing environment. The extent to which an environment can be personalized depends on the technology being used, and it is difficult to predict which will become dominant in the future. Thin client technologies such as *GoToMyPC* [2], *Terminal Services* [31], and *VNC* [26] allow users remote access to their own machines. With *Internet Suspend/Resume*®, an enterprise client management system, a user's computing environment follows the user through the Internet as he or she travels [18, 28]. Still others, such as *Migo* [4], use portable storage devices to allow the user to carry his or her environment settings and personalize a computer "on-the-fly."

The near ubiquitous availability of hardware, new applications, and the ability to easily personalize a computing environment using technologies like those listed above are making the *transient* use of *unmanaged* hardware more common. Unmanaged hardware refers to shared computing devices that a user does not own or administer but may use for short periods of time, such as a loaner laptop at an airport, or a public terminal connected to a corporate network. In contrast, *managed* hardware refers to a machine that is exclusively accessed by a single user or set of authorized users, and administered by the user(s) or a designated administrator. The transient usage model is centered around mobile users, and may involve intervals of short usage in unexpected locations. The ability to personalize hardware eliminates the need for a user to be concerned with the physical location of his environment. As a side benefit, this results in increased user productivity over periods of short usage, because the user is already familiar with the computing environment.

The transient usage model today requires users to accept on good faith that the systems they use do not behave maliciously. Systems designed for transient use often suffer from problems such as poor maintenance, which increase security risks. Keeping these risks in mind, there are three important characteristics to note about this usage model. First, systems are often accessed for very short periods of time. Second, during periods of short usage, the user's *trust footprint*, which comprises the number of applications that the user will access, is likely to be small. Third, since very often systems are unmanaged, users cannot rely on administrators to protect them from attacks. A viable security solution would need to be user friendly and fast enough to satisfy the user's time constraints. We discuss a well studied related problem, the untrusted terminal problem, in a later section.

Most existing solutions proposed in the literature to enhance security rely on system administrators to undertake the task of protecting users from attacks, and do not address the concerns of transient use. In light of this new usage model, methods to give users a higher level of confidence in unmanaged computing systems need to be explored.

In this paper, we describe the Trust-Sniffer system, which addresses the above concerns. Our approach is pragmatic. We focus on software attacks, and do not guard against hardware modifications or virtual machine based attacks. Our work is an attempt to explore the area of achieving maximal security gains while placing minimal requirements on the user, and preserving ease of use.

We use a staged process of helping a user establish confidence in the software stack on an untrusted machine. Initially, a minimal, trusted, user-carried passive device is used to boot the untrusted machine, and help establish confidence in the operating system on the machine. Subsequent integrity checking is accomplished using a modified version of the Linux kernel, which checks application binaries and dynamic linked libraries on demand as they are executed, and prevents them from executing if they cannot be validated. It is important to distinguish our minimalistic use of boot software from approaches such as SoulPad [12], which bring an entire operating system and applications. This is not always practical, and we discuss this in detail in Section 6. We distinguish *execution state* from *disk state*. Applications that are not executed do not have to be checked. An infected binary is not a threat to the user as long as it is not executed. In addition, if an attempt to execute it is made, our system ensures that the user will not be compromised, since untrusted applications are prevented from executing. The **main contributions** of our work include:

- A study of how to significantly improve security without burdening the user.
- A simple, user friendly implementation, based on a small, passive device that does not depend on special tamper proof hardware.
- A mechanism to rapidly validate *user relevant* portions of the software stack on an untrusted machine.
- An efficient system to protect users from the inadvertent execution of malicious code.

In the sections that follow, we elaborate on the staged process of establishing confidence in an untrusted machine. In Section 2, we introduce background information about trusted computing. Section 3 describes the architecture and operation of the Trust-Sniffer system. In Section 4, we describe the implementation of the system. In Section 5, we evaluate the system, discussing its performance, and security guarantees. Section 6 discusses some questions raised by our approach. Finally, Section 7 looks at related work and in Section 8, we suggest future improvements to our system.

# 2 Background

In this section, we give an overview of the methods used by attackers to compromise vulnerable systems. We then provide background information about trusted computing and introduce existing work in the literature that is relevant to our system.

## 2.1 Methods of Intrusion and Code Modification

Computers connected to a network are susceptible to remote attacks where a malicious party exploits a software vulnerability to obtain access to the system. Once the attacker obtains access to the system, he can maintain unauthorized access to the system, for example by installing a kernel-level rootkit. The rootkit could accomplish this by replacing system binaries with malicious ones. It also allows the attacker to execute malicious code undetected by the operating system. Other threats include viruses, worms and malware.

These problems have been commonly studied in the context of managed hardware, and are an even bigger threat to systems maintained by non-expert users. The transient usage model also complicates the intrusion problem, because users typically have physical access to the system. This allows certain attacks to be executed even without network connectivity.

Modifications to the software stack could cause potentially disastrous results, especially when users begin to trust their personal data to unmanaged systems. Such modifications could be as innocuous as a

commonly used system utility, such as `ps`, or as significant as modifying the operating system, such as with a Loadable Kernel Module (LKM). Even without administrative privileges to the system, BIOS permitting, an attacker could boot a shared system with a live CD such as Knoppix [3] and carry out unauthorized modifications to the software on the local hard disk drive. Sensitive data of subsequent users could be compromised by a malicious application that the attacker installed.

### 2.2 Trusted Computing Primitives

Most solutions that protect systems from intrusions involve mechanisms that ensure that the integrity of a system is maintained in the face of attacks. This is accomplished by detecting changes to the software stack on the system and comparing them to a known baseline configuration.

Before we describe our approach, we present two currently proposed mechanisms, *trusted boot* and *secure boot*. However, both require modifications to the platform. The Trusted Computing Group's (TCG) standards specify the use of a secure co-processor, the Trusted Platform Module (TPM) to store sensitive state [30]. Once data is stored in the TPM, it cannot be tampered with. The *trusted boot* mechanism makes use of the TPM to establish confidence in a PC's bootstrap process. It extends a chain of trust from an established unmodifiable base called the *root of trust*. When a machine boots, the BIOS verifies the bootloader by computing a SHA-1 hash of its executable code, and saves this information to the TPM. Following this, each component in the boot process verifies the subsequent component that is loaded and extends the TPM with this information. A remote party can then attest the integrity of the boot process using a challenge response mechanism to obtain the state stored in the TPM.

The *secure boot* approach starts with a root of trust which is the initial BIOS bootstrap code. Before loading each subsequent piece of software, the current component verifies the digital signature of the next component [9]. If the digital signature of a component is incorrect, it is prevented from executing and the boot process is halted. The secure boot procedure does not make use of any special hardware, but it requires modifications to boot components, notably the BIOS, which needs to contain signature verification code. The significant distinction between the two approaches is that with trusted boot, the TPM only provides *post-facto* discovery of anomalies in the software stack, whereas secure boot protects the system from malicious code by preventing its execution. In addition, although secure boot prevents the execution of malicious code, it does not provide a way for a third party to verify the code running on the system.

### 2.3 Integrity Measurement Architecture

Our system builds on the implementation of an Integrity Measurement Architecture (IMA) for Linux [27]. IMA is an instantiation of the trusted boot process, and uses the TPM to store system integrity measurements.

IMA's main goal is to build a mechanism which can be used to establish trust in a system, with the aid of the TPM. The integrity measurement process involves computing a SHA-1 hash over the contents of an executable when it is loaded. This SHA-1 hash is referred to as a *measurement*. IMA does not assume that the level of integrity provided using such a method guarantees that programs are completely invulnerable to attack. A program's static fingerprint at load time does not mean its behavior cannot be altered at run time. For example, programs are often modified at run-time using configuration files, which is another avenue for attackers to exploit program vulnerabilities.

IMA is built into the kernel, and is invoked as necessary when any executable is loaded including user applications and kernel modules. Each loaded executable is measured and its hash value stored in a list inside the kernel. Each measurement also extends a predefined Platform Configuration Register (PCR) in the TPM. In this case, the TPM serves a dual role: (i) allowing a remote challenger to verify integrity information, and (ii) preventing malicious software on the machine from altering the measurements or evading

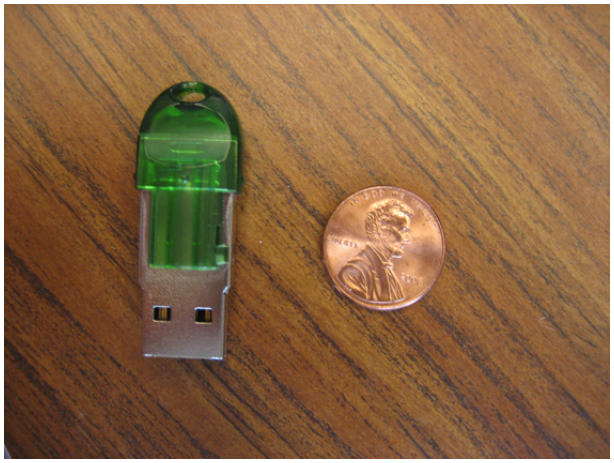

Figure 1: The trust initiator device is convenient for users to carry around. This picture shows a small USB memory stick that is part of a pen.

the measurement system. Since information once stored in the PCRs of the TPM cannot be removed, a malicious application cannot cheat by altering its measurements after they have been stored into the TPM. However, since IMA is part of the kernel, trust must be established in the BIOS and bootloader before integrity verification responsibilities are handed off to the kernel. We discuss this in further detail in Section 4. Finally, IMA does not interfere with normal system operation – it does not prevent untrusted software from executing, and simply provides a mechanism for a remote party to verify system integrity. Our system builds on IMA to suit the model of relegating security decisions to users. Since remote party attestation is not the intent of our system, we do not require special hardware such as the TPM. We use a small, user carried passive device to initiate the process of establishing confidence in the bootloader and kernel on the untrusted machine. The OS kernel on the machine is then responsible for measuring software applications and preventing untrusted software from executing.

# 3 Trust-Sniffer

The goal of Trust-Sniffer is to enhance security with minimal user effort. In the following section, we give an overview of the system, its assumptions and the attacker model. We later provide a high-level description of the individual steps necessary to establish trust in an untrusted machine.

## 3.1 System Overview

The Trust-Sniffer system is intended to increase user confidence in an untrusted machine. We design the system to facilitate the transient usage model, and accomplish this by validating only the software needed for the user's task. By validating the untrusted machine's software stack, a user can take advantage of preconfigured settings in the local environment, such as those for printers and network connectivity.

The system is based on a minimal device, a user carried USB memory stick. From a user perspective, this *trust initiator* device has two major advantages: (i) its small size and passive nature make it inexpensive, easy to replace and convenient to carry around (see Figure 1); and (ii) the process of establishing trust in an untrusted machine is based on a trusted physical possession.

The trust initiator is responsible for establishing a root of trust on the untrusted machine, and enabling the on-disk kernel with the necessary tools to validate application software. The trust initiator contains a list of SHA-1 hashes that is transferred to the on-disk kernel, which uses the hash list to validate application software. Establishing trust in an untrusted machine is summarized in the following steps:

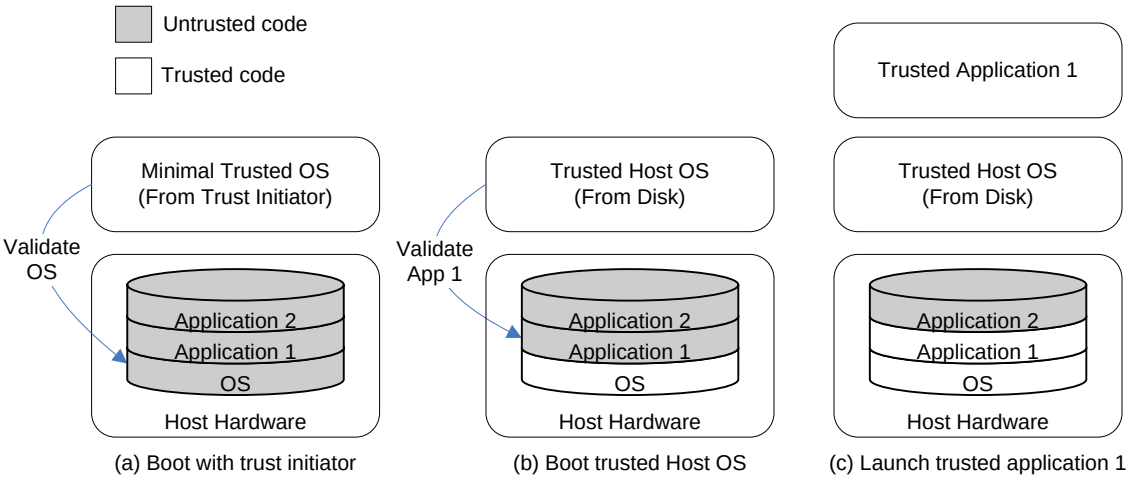


Figure 2: Overview of the staged process of trust establishment.

1. Booting the untrusted machine using software on the trust initiator, and establishing a root of trust by validating the on-disk operating system.

2. Delegating trust establishment in application software to the validated and thus trusted operating system on the machine.

3. Booting the on-disk operating system, which validates applications as they are accessed, and prevents their execution if they cannot be validated.

4. Alerting the user when trust in an application cannot be established.

This staged process is illustrated in Figure 2.

### 3.2 Threat Model and Assumptions

The primary goal of our system is to protect the user from modifications to the system software stack. The consequence of compromised software is a potential disclosure or loss of integrity of sensitive user data. Since most machines today are connected to a network and in particular, physical access to unmanaged hardware is generally not restricted, attackers can modify applications or instrument the system with key logging or screen capture software, or a kernel-level rootkit.

In order to use our system, we assume that the user is allowed to reboot the untrusted machine, and the BIOS allows booting from a USB memory stick. We also assume that modifications to the BIOS are difficult. Hence, we do not guard against virtual machine based attacks, where a compromised BIOS could boot directly into a malicious virtual machine monitor [17]. We also assume that the operating system kernels on untrusted machines are equipped with appropriate software to carry out program validation. We describe the software requirements in detail in Section 4, and ways in which this assumption can be relaxed in Section 5.

We observe that the most common attacks are software attacks, and hence do not guard against hardware attacks, which could be prevented by tamper proof hardware and physical surveillance. We also do not consider physical attacks such as “shoulder surfing,” which can be addressed using products such as screen protectors.

Since our system is based on *load-time* program validation, it does not protect against run-time attacks. Specifically, IMA allows programs with valid checksums to load and execute, but if an attacker is able to execute an attack at run-time, the attack would go undetected. This is referred to as a Time-of-Check-Time-of-Use (ToCToU) violation. To achieve stronger security guarantees, systems that provide run-time attestation, such as Pioneer, can be used [29].

## 3.3 Operation

Establishing trust in an untrusted machine consists of three steps. The Trust-Sniffer system (see Figure 3) consists of three major components: (i) the software components on the trust initiator device; (ii) the Trust-Sniffer kernel module; and (iii) a user space notifier application. In the following section, we describe the role of each component in relation to the steps needed for trust establishment.

### 3.3.1 Rapidly Establishing a Root of Trust

Other than the BIOS, initially, none of the software on the machine is trusted. The goal of the trust initiator is to (i) bootstrap the trust establishment process by establishing a root of trust; and (ii) equip the on-disk OS with necessary tools to validate the rest of the software on the machine. These tasks are accomplished as rapidly as possible to minimize system down time.

The trust initiator is equipped with appropriate software to boot the untrusted machine. Once the machine is booted, the trust initiator's bootstrap software accesses the machine's disk to validate the integrity of the kernel image, and any other components that would be required during boot, such as initial ramdisks. The OS and related boot components serve as a *static* root of trust, since as described previously, we perform load-time validation. In contrast to *attestation* where untrusted software is discovered after it has been loaded, *validation* refers to detection of untrusted software before it is used.

The software in the root of trust turns out to be relatively small, and the validation process does not significantly increase the time needed to boot the system. The trust initiator contains a list of measurements of the software comprising the root of trust. If any component in this subset does not match its expected measurement, the trust establishment process is halted. If allowed to continue, the system would not be able to guarantee the integrity of any other portion of the software stack, because trust in the OS could not be established.

### 3.3.2 Dynamically Extending the Root of Trust

Once the core root of trust has been established, the kernel takes over the task of extending trust to applications. As described in previous sections, IMA maintains an ordered list of measurements in the kernel of applications as they execute. This measurement list and its aggregate hash stored in the TPM allow a remote party to attest the software running on the system using a challenge response mechanism. Our system does not perform remote attestation and hence does not strictly require the measurement list to validate applications. As we describe below, when an application is measured, its measurement is validated in place and can then be discarded. However, we retain the in kernel measurement list mostly for auditing and informational purposes.

The Trust-Sniffer kernel module, known as the *trust extender*, enhances the IMA implementation by performing validation of the software in place, instead of attestation by a remote party. When an application is measured, its measurement is compared to a list of known trusted measurements. If the trust extender cannot validate an application, it blocks the application's execution. In order to accomplish in place measurement validation, the kernel needs to be made *trust aware*. The trust initiator device contains the expected list of measurements for the software stack, and transfers it to the kernel before it is booted while establishing the root of trust.

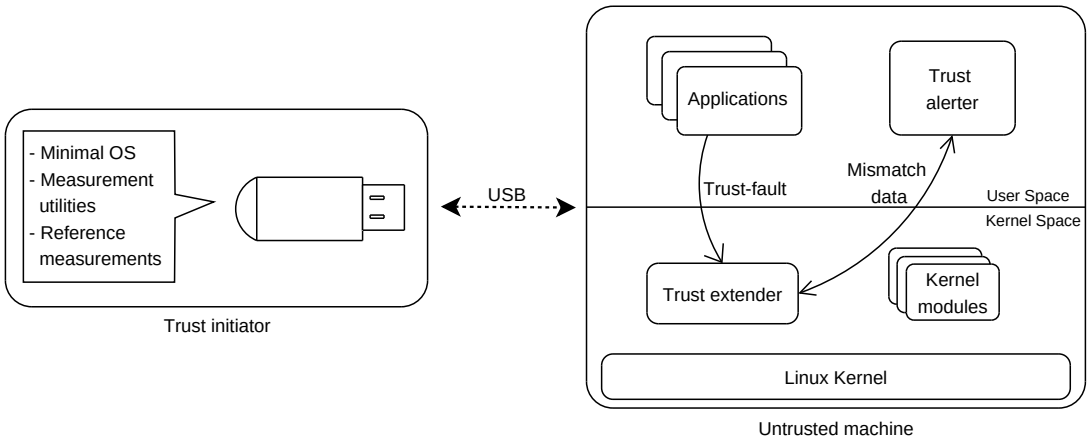


Figure 3: Trust-Sniffer architecture. The trust initiator plugs into an untrusted machine, validates its OS, and equips it with reference measurements. Applications that have not been validated cause a trust fault, which is handled by the trust extender. If an application's measurement is not recognized, the mismatch is forwarded to the user space trust alerter.

Once the kernel has booted, it becomes the sole trusted agent responsible for validating the software stack. This has several benefits. There is no need to establish secure communication with any other party, and secure hardware is not required because the kernel is already trusted. It also facilities the goal of rapid authentication. Applications are only verified when they are accessed. If a user only uses a small percentage of applications on a disk, this can result in significant time savings. When a new (untrusted) application is executed, the trust extender handles a *trust fault* and attempts to validate the measurement. The decision on whether to give the application permission to run is cached to speed up subsequent executions.

### 3.3.3 Handling Untrusted Applications

Since the Trust-Sniffer system is designed for end users, there needs to be a mechanism to inform them when an untrusted application is detected. When the measurement of an application cannot be verified, the kernel prevents the application from executing. The user is not aware of communication between applications and the kernel. Hence, without the kernel providing additional information about the application's execution state to the user, it would appear as an unexpected failure.

The Linux kernel has a built in error handling mechanism so that applications can recover from system call failures. This is implemented in the form of error codes, where each code can indicate a different type of failure. An application must be equipped to handle errors and take the appropriate action, such as informing the user of the failure in an interactive application. However, in the case of an untrusted application measurement, using the application to communicate failure information to the user is no longer an option, since the application's execution is blocked. To solve this problem, the Trust-Sniffer system is equipped with a user mode listener application, the *trust alerter*, that communicates with the trust extender. When a failure occurs, the trust extender transfers relevant information, such as the application that failed and reasons for the failure, to the user mode listener, which can then be communicated to the user.

## 3.4 Example Use

We describe the following example to illustrate the use of our system. Bob is at a public computer at the airport wants to edit a document. He plugs in the trust initiator. The initial scan quickly indicates that the operating system is safe. After the machine boots, he launches a word processing application. The application is validated in the background, and since it is determined to be safe, the system allows the

application to execute. Alice comes along, and wants to use the same machine to check her email. The trust initiator works as before. She launches a web browser. Its measurement is found to be suspect, and the system indicates to Alice that the application she attempted to launch is unsafe. Since Bob's trust footprint did not comprise a web browser, the presence of an unsafe web browser did not prevent his document edits. For Alice, who has a different trust footprint, the system finds an untrusted application and prevents its execution. Note that applications may have complex dependencies, such as dynamic loadable libraries. We describe how the system handles different executables in the next section.

# 4 Implementation

In this section, we describe the details of the software components on the trust initiator, and how the root of trust is established on the untrusted machine. The modifications we have made to the Linux kernel include a kernel module, the trust extender. This module builds on the IMA implementation, and leverages the Linux Security Module (LSM) interface to validate executables. We also describe the details of the interaction between the trust alerter and the trust extender.

## 4.1 Experimental Setup

Our prototype implementation was carried out on a Fedora Core 5 distribution using version 2.6.15 of the Linux kernel. Our test hardware consisted of an IBM T43 laptop with a 2.0 GHz Pentium M processor and 1 GB of RAM. We use a standard USB memory stick with 1 GB of storage capacity as the trust initiator, although 128 MB would have been sufficient for our purposes.

## 4.2 Trust Initiator Preparation

An important point is that IMA does not measure any of the software loaded during boot. The implementors refer to a mechanism to validate the boot process described in [19]. When the kernel boots, it expects measurements of itself, the BIOS, and the bootloader in a predefined PCR of the TPM [27]. The initial boot measurements can be made using *TrouSerS* [7], an open source trusted bootloader and a generic TCG based software stack. If the aggregate measurements of the software loaded during the boot process differ from their expected measurements, measurements taken by IMA are not reliable, because the kernel could be compromised. In the Trust-Sniffer system, we make use of the trust initiator's boot to validate the kernel.

In order to make the USB stick bootable, it needs to be partitioned, formatted and loaded with a bootable operating system. Partitioning and formatting USB memory sticks depend on the vendor and the drive geometry[1], and the details are left out here. We format the USB device with the FAT filesystem, and use the Finnix [1] distribution as the operating system. Finnix is a descendent of Knoppix and has excellent support for devices and automatic hardware detection. It is more suitable for our purposes than Knoppix because it is a stripped-down distribution intended for administration purposes, it boots faster and has a small footprint. It also has native support for Logical Volume Management (LVM) [32], which many distributions, including Fedora Core, use to manage storage. The Finnix ISO image is less than 100MB, so a memory stick with 128MB would be sufficient, as mentioned earlier.

The trust initiator also needs to be loaded with *reference* measurements for the initial boot validation phase as well as subsequent application validation by the kernel. To avoid confusion, we refer to measurements taken by the Trust-Sniffer as *sample* measurements, which are checked against a list of reference

[1] Although the geometry of a USB memory stick is not physically comprised of cylinders, heads and sectors, the stick is encoded with logical CHS information. Older BIOSes used CHS as the mode of addressing drives, newer advanced BIOS implementations use logical block addressing.

| **User** | **Trust Initiator** | **Trust Extender** | **Trust Alerter** |
|---|---|---|---|
| wants to use $App_i$ | possesses *RM* | | |
| inserts *TI* into untrusted machine | | | |
| | validate(*OS*) | | |
| | boot *OS* | | |
| | $\xrightarrow{RM}$ | | |
| | | validate(*Init*) | |
| launches $App_i$ | | | |
| | | validate($App_i$) | |
| | | if $M(App_i) \in$ *trusted list* | |
| | | run($App_i$) | |
| | | else | |
| | | block($App_i$) | |
| | | $\xrightarrow{Mismatch}$ | |
| | | | alert user |

Figure 4: Information flow across the system. When the OS boots, failure to validate all measurements of programs launched during *Init* causes the kernel to panic. This is omitted for simplicity. RM = Reference Measurements

measurements. An inequality between a sample measurement and every reference measurement in the list is called a *mismatch*. Note that a sample measurement that is equal to any reference measurement can be trusted, since reference measurements are generated from trusted applications. Reference measurements are obtained from a trusted machine by scanning the entire disk and recording every measurement. The reference measurement list only has to be generated initially and updated when patches and new software are released. 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 our reference list contains measurements for the latest version of an application. However, it is conceivable for the list to contain multiple trusted measurements for an application such as the kernel. In this case, if a system with multiple kernels was encountered, the system could permit the execution of a trusted kernel even if the remaining ones were compromised. To address the problem of frequent software updates by vendors, users could obtain new measurements from a server that periodically generated updated lists and digitally signed them for distribution. We discuss this issue further in Section 6.

## 4.3 Staged Validation of Measurements

### 4.3.1 Validating the Boot Process

A custom startup script on the trust initiator mounts local hard disks and discovers the boot partition. It then uses `sha1sum` to measure the kernel image, the initial ramdisk, and GRUB, which Fedora uses as the bootloader. As described earlier, the trust initiator contains reference measurements for this set of software. A few custom utilities are used to compare the measurements of this set of software to the reference measurements on the trust initiator. Any mismatches would cause the boot process to be untrusted, and the user is then informed of whether he would like to continue using the system.

#### 4.3.2 Transferring Control to the Kernel

Before booting the on disk kernel, it needs to be made aware of the reference measurement list. The trust initiator copies the list to a predetermined location on disk. We avoid the user having to do a manual reboot (powering off the machine, removing the trust initiator, and then powering the machine on) by using the `kexec` utility, which allows replacement of the current running kernel without using a bootloader [5].

The kernel module uses the `securityfs` pseudo filesystem to communicate with user space programs. This file system is used to allow user space programs to read sample measurements, and also provides an interface for programs to issue measurement requests. We extend this with a new interface to allow a user space program to load the trust extender with the list of reference measurements.

One important point to note is that the trust extender can only begin enforcing measurement mismatches after the reference measurement list has been loaded. It is thus imperative that this list be loaded as early in the boot process as possible. Until the list is loaded, the trust extender is non intrusive, and allows all execution. The trust initiator in part addresses this problem by validating all boot time measurements. We use a custom initrd image with modified boot scripts that launches a user space utility to load the reference measurements into the trust extender. The list loading utility is preconfigured with the location of the reference measurement list on disk. Since all initial execution before the list is loaded is from the initrd, software is allowed to execute. Once the reference measurements are loaded, the trust extender disables the interface in `securityfs` that allows this, to maintain integrity of the list. At this point, it also begins actively disallowing applications with mismatched measurements from executing.

#### 4.3.3 Updates to the Kernel Measurement Mechanism

The kernel measurement mechanism implemented by IMA accounts for user level executables, dynamically loadable libraries, kernel modules and scripts, which are the ways in which code can be loaded and executed. The measurement procedure is described in detail by Sailer et al. [27], and we refer to relevant parts below. Note however that we do not use the TPM in our implementation.

The reference measurement list is stored in a hashtable indexed by the SHA-1 hash value of each executable. We do not store the path names of executables, because this is irrelevant in the context of the measurement process. Pathnames are not a reliable way of identifying files. In addition, if the measurement of a file matches a reference SHA-1 value, its integrity is established and the file's location on the filesystem has no bearing on the security of the system.

The trust extender uses the `file_mmap` LSM 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 SHA-1 hash value of the executable code, and some additional file metadata such as pathname, user ID and group ID is also stored. The metadata does not affect the actual measurement validation procedure, but we use it to communicate information to the user. This is described in the next section. When a measurement is taken, its value is compared against the reference list. If there is no matching reference measurement, the measurement is considered untrusted, and the kernel disallows the execution. All measurements are recorded in the kernel into a list. We no longer need the list for remote attestation, but retain it for consistency and auditing purposes.

A caching mechanism is used to reduce the overhead of measuring unchanged applications in the case where they are executed more than once. Each measurement is tagged as `CLEAN`, `DIRTY`, or `MISMATCH`. The tagged value is stored in a special security substructure of the inode datastructure in the kernel, and is also part of each entry stored in the measurement list in the kernel. When a file that was previously measured is executed again, the kernel simply needs to examine the inode value to see if the measurement needs to

be recomputed. A value of CLEAN indicates that the measurement is trusted, and MISMATCH indicates that the measurementis untrusted. DIRTY indicates that the measurement is stale and needs to be recomputed. A measurement is flagged as dirty when the file is opened for writing, or if it is on a filesystem that is unmounted.

#### 4.3.4 Communicating Information to the User

There are two instances when communicating information to the user is necessary: (i) when the trust extender encounters an untrusted application; and (ii) when a failure occurs in the kernel which would cause the measurement process to become compromised. The second case encompasses unexpected failures in the measurement process, such as out of memory errors.

To establish a channel for communication between the trust alerter and the trust extender, we use the netlink socket interface. We install a new netlink protocol type, TS_NETLINK and add the TS_NLMSG_MISMATCH as well as 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 is launched as the system boots, and waits for messages from the kernel using the recvmesg system call. When an untrusted application is encountered, it sends the appropriate message with a call to netlink_broadcast. In the case of a mismatch, this includes the pid of the process that failed, and the filename of the application on disk, if available. We have implemented a simple console application that communicates with the kernel in the current implementation. This can be converted into a more friendly graphical user interface application in future iterations of the system.

## 5 Evaluation

In this section, we evaluate Trust-Sniffer in light of user expectations about performance, and the security guarantees provided by the system. The discussions in this section serve to revisit the choices we made in designing our system and assess their effectiveness in the system's overall usability, security and extensibility.

### 5.1 Security

In accordance with design goals we presented in earlier sections, the system needed to strike an appropriate balance between ease of use and security guarantees. We sacrifice some security guarantees for the sake of simplicity, observing that very often complex solutions are impractical for non-expert users.

We acknowledge that Trust-Sniffer is not foolproof and is vulnerable to certain attacks. First, our system is vulnerable to user error. If the user forgets to reboot the untrusted machine, the system cannot provide any security guarantees. Second, related to rebooting the machine, we do not guard against a malicious BIOS or virtual machine monitor, which could both provide the illusion that the machine was booting from the USB device even if this was not actually the case. Third, some local configurations could cause increased security risks. For example, if a machine was configured such that several ports were left open and the firewall disabled, our system would not be able to detect that the configuration was malicious. Finally, our system is vulnerable to physical attacks.

In spite of these specific vulnerabilities, we believe that the guarantees provided by our system are a substantial improvement over the precautions that users take today. Our system pessimistically disallows the execution of unknown software (i.e. software for which there is no reference measurement present), which may inconvenience some users. We believe advantages of increased security outweigh some user inconvenience in this context, since encountering an unknown application is not the typical case. The reference list can easily be equipped to contain measurements for the most commonly used applications.

```
[asurie@shackleton]$ xterm
Killed
[asurie@shackleton]$
```

(a) Execution of an untrusted application is terminated

```
[root@shackleton]$ ./trust_alerter
Trust Alerter
The application xterm is untrusted and cannot
be validated.  Execution of process with Id 3535
has been blocked to minimize security risks.
```

(b) Notification to the user from the trust alerter

Figure 5: User experience when an untrusted application is encountered by the trust extender.

Since the trust initiator is write protected, its software cannot be modified by unauthorized external sources. Of course, a user can disable write protection on his own machine to allow the reference measurement list to be updated. In addition, the trust establishment procedure does not depend on network communication with the untrusted machine; thus we disable networking capabilities on the minimal OS that we use to validate the on-disk kernel.

To demonstrate the ability of the system to prevent untrusted code from running and compromising the software stack, we run an experiment. We generate a measurement list snapshot of the system. Before loading the kernel with this measurement list, we alter the reference measurement of a common application, such as `xterm`. Once the kernel has been setup, we test the execution of `xterm`. The results of this execution in a shell as well as feedback from the kernel are shown in Figure 5.

This experiment shows that our system should be useful in detecting and informing the user about common occurrences, such as applications which have not been patched, or unrecognized applications which the user should not trust.

## 5.2 Performance

From the perspective of the transient usage model, we evaluate how much overhead is required by the Trust-Sniffer system in comparison to a system without security checks. The metrics that matter most to the user are the time the system requires to boot, as well as the ongoing overhead required to check applications. There are two important aspects of this evaluation to keep in mind. First, in a system without security guarantees, a user would not typically be required to reboot the system in order to use it. However, we illustrate below that with the Trust-Sniffer, this downtime is minimal and the additional security is worth the performance penalty. Second, the performance overhead for application measurements is only on the first execution.

An evaluation of the overhead of measuring applications has been done by Sailer et. al [27]. However, we do not have to perform expensive operations to extend measurements to the TPM, and our implementation allows the kernel to do measurement validation in place. On account of these differences, we evaluate the overhead in the context of our system. Since the trust extender primarily uses the `file_mmap` LSM hook to measure applications at load time, we measure the latency of an `mmap` operation in different contexts using the HBench-OS framework [11]. Table 1 shows the latency of a trust fault.

When an application is executed for the first time, its SHA-1 measurement is computed and we denote this as a trust miss. If on subsequent executions an application's cached measurement is valid, its execution

| Type | Latency (stdev) | Overhead |
|---|---|---|
| Trust Hit | 1.19 $\mu s$ (0.07) | 0.20 $\mu s$ |
| Trust Miss | 4.29 $\mu s$ (0.06) | 3.3 $\mu s$ |
| Reference | 0.99 $\mu s$ (0.04) | – |

Table 1: Latency of an `mmap` operation with the trust extender.

| Configuration | Boot time (stdev) | Overhead (percent) |
|---|---|---|
| Trust-Sniffer | 111.4 *sec* (0.52) | 14.3 *s* (14.2) |
| Standard | 97.1 *sec* (0.57) | – |

Table 2: Comparison of the time taken for the system to boot

results in a trust hit. The terms trust hit and trust miss refer to the system's cache of measurements it has already seen, and do not denote whether the application is trusted or untrusted. The reference data indicates the latency of an `mmap` operation without the Trust-Sniffer. Note that the SHA-1 operation is performed only in the case of a trust miss.

It is clear in the case of a trust hit, the overhead of validating an application that has previously been measured is not significant. When an application is first executed, the measurement overhead is high compared the reference, however, this is done only once.

To evaluate the boot process, we measure the average time it takes to boot a machine from the point the power button is pushed until a login prompt appears. This metric is relevant to the user, since it defines the "warm-up" time needed by the system before the user can start doing work. Table 2 shows these measurements. The standard configuration refers to a system without Trust-Sniffer. The overhead of using Trust-Sniffer is minimal, only 14.2% over a standard system. One might argue that typically users are not required to reboot a system. However, we believe that this additional step should not inconvenience users in practice; the additional 2 minutes spent to boot an untrusted machine should be well worth the resulting gains in security.

Finally, as a point of discussion, we explore ways to improve the performance of the kernel measurement mechanism. One observation we have made is that there are likely to be performance gains if it is possible to reduce I/O overhead in reading a file to be measured. Currently, the measurement of an application is the SHA-1 hash of all of its executable code. If an application makes calls to any shared libraries, which is very often the case, the libraries also need to be loaded into memory and measured.

The Linux kernel uses demand loading for executables, and when an application is executed, the required code is paged into memory on demand. For large applications, it may be the case that only a small portion of the executable code is mapped. This is true of library calls that the application makes, because typically an application does not require all the functionality of a given library. One way to reduce the I/O overhead of measuring applications could be to use a hash-tree based scheme to validate only the executable code of a file that is loaded. Under this scheme, the measurement of an application would be broken up into a series of individual hash values based on a predetermined block size, and measurements could be validated at a finer granularity. To accomplish this, the kernel measurement mechanism would have to be updated to hook into the page fault handler, so that only code that was actually executed would be measured. Although this approach looks promising, its feasibility and security implications need to be investigated.

### 5.3 Usability and Extensibility

Given an increasingly broad user base, an important trend in computing is to minimize the requirements placed on the user, and the Trust-Sniffer addresses this concern from a number of different angles. The user needs to carry only a simple device that can dramatically increase user awareness and security for simple day to day computing tasks. In addition, we attempt to draw similarities between computer security and how users view physical security. In the lock and key model typical of physical security, the trust initiator can be likened to the lock, since through the measurement list it determines which applications are permitted to execute. This "lock" serves to protect user data from malicious applications.

Our design is flexible and extensible. As suggested earlier, it would be easy to set up a mechanism for users to obtain updates to reference measurement lists. The implementation is based on a specific distribution, but it would be easy to adapt the trust initiator to hook into the boot procedures of other Linux distributions. In addition, aside from the custom hook to monitor the loading of kernel modules, the in kernel portion is based entirely on the LSM interface. As the LSM interface expands a kernel module hook could be included in the future. It would then be possible to compile the trust extender as a kernel module for stock kernels of various distributions. The trust initiator would then be able to configure arbitrary machines on demand, enabling the use of machines without kernels preconfigured for the Trust-Sniffer system.

## 6 Discussion

With the trust initiator, a user essentially carries a minimal operating system on a USB memory stick. In this regard, one might think whether it would be more useful to bring a full operating system and applications on the USB stick (or on CDROM/DVD, although these media types are mostly read only), which would eliminate the need for establishing trust in the software stack on an untrusted machine. More sophisticated systems such as SoulPad [12] and Personal Server [33] provide a way for a user to carry his personal computing environment (including applications and data) with him on a mobile device. To avoid ambiguity below, let us refer to this as portable software.

There are a few reasons why this may not always be practical. First, portable software configurations would not contain correct local environment configurations such as printer and network settings that would be needed for correct operation. Second, although automatic hardware configuration is improving and a large number of generic drivers are bundled with operating systems, it is not possible to guarantee that a portable OS will have every driver necessary for the hardware on an unknown system. This might prevent the user from being able to use common peripherals or devices. With Trust-Sniffer, having the necessary drivers on the trust initiator is not as big a concern. Establishing the initial root of trust requires only a basic set of hardware on the machine. It is assumed that as long as the trust initiator software is reasonably up-to-date, common devices such as disks should be well supported. Network and wireless cards are not required to be operational for the trust establishment process (and they are often the devices for which specific drivers are not bundled with operating systems). Third, very often individual users do not have licenses for certain applications. In such cases, relying on portable software is not an option. Our observation in light of these reasons: boot the system at hand with its local configuration, but make sure that all executables are trusted. Obviously some local configuration settings could be malicious, but this is not the common case.

Keeping the software on the trust initiator updated should not be difficult for users. Since the boot software is very minimalistic, it should be easy to patch and update as new versions are released. Also, the list of trusted measurements has to be updated as vendors release patches or new versions of software. This includes the addition of new measurements, and the purging of old measurements that are no longer trusted. One could imagine an infrastructure to facilitate this process, where software vendors (or other

trusted entities) would provide secure servers from which updated measurement lists could be obtained. Users could then update the trust initiator by simply running an application that obtained the list from a trusted entity.

Another point of discussion is that since the Trust-Sniffer system is not foolproof and does not offer perfect security guarantees, users may get a false sense of security. In particular, the security of the system relies on the trust initiator being updated regularly, which may make it vulnerable to zero-day attacks. The update problem is no different from the use of personal desktop operating systems today. If a user does not patch a security hole immediately (or the software vendor does not release a timely patch, which is not unusual) a zero-day attack is still possible. Compared to usage of untrusted systems today where users take no precautions at all, Trust-Sniffer dramatically increases security without placing a burden on the user. In addition, the process of using our system makes users conscious of risks they face and increases overall awareness about security.

# 7 Related Work

When comparing the Trust-Sniffer system to other work, we consider the following: (i) most systems are designed with an administrator in mind and are not focused on providing security guarantees to users; and (ii) although many solutions offer more security than our system, they have complex hardware and software requirements, which slows their adoption in practice.

A well studied problem is the untrusted terminal problem. This is the problem of a user interacting with an untrusted device, and the most interesting formulation of the problem is that of using an untrusted device to establish secure communication with a trusted remote device. Clarke et al. propose camera-based authentication to establish an authenticated bidirectional channel between the remote device and a user on the untrusted terminal [13]. Other solutions that rely on securing the untrusted terminal include Naor and Pinkas' visual cryptography approach to provide authenticity and secrecy of messages displayed on the terminal [21], and the trusted smart card approach by Abadi et al. to delegate authority to the untrusted terminal without revealing user credentials [8]. Gobioff et al. also propose a smart card based solution [15]. A number of solutions do not rely on the use of the untrusted terminal hardware, and instead use trusted hardware carried by the user, such as a PDA like device [10, 13, 24, 25].

The secure boot mechanism, described earlier in Section 2, allows validation of the boot process, with the benefit that untrusted code is prevented from executing [9]. As a system is booted, each component in the boot process validates the next component before it is loaded to establish a chain of trust. However, this procedure requires modifications to the BIOS, and all software modules loaded during the boot process.

The Tripwire project provides software monitors and audits changes to specific files from a known baseline configuration [6]. It detects changes to files by analyzing periodic snapshots of the file system. This solution is beneficial to system administrators, who often need to maintain standard system configurations for their users, and is not intended to provide end users with any guarantees on their personal data. In addition, its security guarantees are limited, because changes are not detected in real time. This means that an attacker could go undetected by compromising the system during an interval when the change detection agent is not running.

Kennel and Jamieson describe methods to remotely establish whether the hardware and software of a computer system are *genuine* [16]. The basic idea is for a remote system to pose a set of challenges to the client, which the client should only be able to solve satisfactorily if its hardware and software match with what the remote system expects. Using such a challenge-response protocol, a remote service, such as an NFS server could refuse access to a compromised client.

Terra is an architecture for trusted computing based on virtual machines [14]. It aims to allow applications with different security requirements to run side by side on the same system in different virtual machines. The core of the platform is based on a trusted virtual machine monitor, which is responsible for allocating hardware resources among virtual machines.

Petroni et al. have developed Copilot, a coprocessor based system to detect modifications to a host operating system's kernel [23]. The Copilot software runs on a separate PCI card which monitors the kernel, without relying on its processor or memory capabilities. This allows detection of rootkits, and other modifications to the kernel. The system is not designed for mobile use, and does not monitor the integrity of applications.

The Pioneer system uses a challenge response protocol initiated by a *dispatcher* to provide verifiable code execution on an untrusted platform [29]. The dispatcher detects malicious tampering by measuring the response time of a verification function that runs on the untrusted platform. Pioneer allows allows for an attacker to have complete control over the software on a system, including administrative privileges, but makes several assumptions, notably that the dispatcher knows the hardware configuration of the untrusted platform in advance. However, since it establishes a dynamic root of trust, it can be used as a building block for future iterations of the Trust-Sniffer.

The Next-Generation Secure Computing Base (NGSCB) developed by Microsoft aims to provide an architecture for isolating trusted applications from untrusted code [22]. NGSCB proposes that sensitive applications run in a special secure mode of the operating system with access to a limited portion of the OS API. Input data from the keyboard, mouse and other devices is encrypted and subsequently decrypted by the *Nexus*, which can then pass it on legacy OS applications or to a trusted application running in Nexus-mode. NGSCB relies on secure hardware, such as a TPM chip, and also does little to handle untrusted applications.

McCune et al. propose BitE, a system to secure sensitive user input from malicious user space applications on an unknown system [20]. The user carries a trusted active device (a cellphone), that communicates with the unknown system to attest its software. Users enter sensitive input using the trusted device, which is communicated to individual applications over a secure channel. BitE differs from Trust-Sniffer in that is not intended to validate applications, and requires a TPM chip.

# 8 Future Work

In later iterations of our system, giving the user the ability to audit when untrusted software is detected could be very useful. One way to accomplish this would be to store this information on the USB stick (integrity of the software on the USB stick could be maintained by make it only writable by the kernel). This could help gather statistics on how often users are exposed to identifiable security risks. The information that is communicated to the user can be enhanced beyond simple information about which application failed a measurement check. Such communication could incorporate what the user was doing on the system when the untrusted application failed, and provide use context relevant feedback. For example, if the measurement of the `emacs` text editor was found to suspect, the system could suggest alternative trusted software, such as `gedit`.

# 9 Conclusion

Trust-Sniffer is a system to help users establish confidence in untrusted machines. This paper explores the security gains that can be achieved with Trust-Sniffer while placing as little burden as possible on the user. We leverage existing user experience with placing trust in physical possessions to build the system around a trusted user-carried passive device. The user device is small and inexpensive, and the system does

not require any special hardware. Although we sacrifice some security guarantees to achieve simplicity and ease of use, overall security for users is greater than it is today. Providing the user with a simple metaphor for a complex problem also helps increase user awareness about the security risks they face.

## Acknowledgements

We would like to thank Reiner Sailer of IBM Research for his assistance with the Integrity Measurement Architecture. We would also like to thank Jan Harkes for his suggestions on the project. Internet Suspend/Resume is a registered trademark of Carnegie Mellon University. All unidentified trademarks are the property of their respective owners.

## References

[1] Finnix. `http://www.finnix.org`.

[2] GoToMyPC. `http://www.gotomypc.com`.

[3] Knoppix. `http://www.knoppix.net/`.

[4] Migo. `http://www.migosoftware.com`.

[5] Reboot Linux faster using kexec. `http://www-128.ibm.com/developerworks/linux/library/l-kexec.html`.

[6] Tripwire. `http://sourceforge.net/projects/tripwire/`.

[7] TrouSerS. `http://trousers.sourceforge.net/`.

[8] ABADI, M., BURROWS, M., KAUFMAN, C., AND LAMPSON, B. Authentication and delegation with smart-cards. *Science of Computer Programming 21*, 2 (Oct. 1993), 91–113.

[9] ARBAUGH, W., FARBER, D., AND SMITH, J. A secure and reliable bootstrap architecture. In *Proceedings of IEEE Symposium on Security and Privacy* (May 1997), pp. 65–71.

[10] BALFANZ, D., AND FELTEN, E. W. Hand-held computers can be better smart cards. In *Proceedings of the 8th USENIX Security Symposium* (Washington, D.C., USA, Aug. 1999), USENIX.

[11] BROWN, A. B., AND SELTZER, M. I. Operating system benchmarking in the wake of lmbench: a case study of the performance of netbsd on the intel x86 architecture. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 1997), ACM Press, pp. 214–224.

[12] CÁCERES, R., CARTER, C., NARAYANASWAMI, C., AND RAGHUNATH, M. Reincarnating pcs with portable soulpads. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2005), ACM Press, pp. 65–78.

[13] CLARKE, D., GASSEND, B., KOTWAL, T., BURNSIDE, M., VAN DIJK, M., DEVADAS, S., AND RIVEST, R. The untrusted computer problem and camera-based authentication. In *International Conference on Pervasive Computing* (2002).

[14] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *In Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[15] GOBIOFF, H., SMITH, S., TYGAR, J., AND YEE, B. Smart cards in hostile environments. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce* (Oakland, California, Nov. 1996), USENIX.

[16] KENNELL, R., AND JAMIESON, L. Establishing the genuinity of remote computer systems. In *Proccedings of 12th USENIX Security Symposium* (2003), pp. 295–310.

[17] KING, S. T., CHEN, P. M., WANG, Y., VERBOWSKI, C., WANG, H. J., AND LORCH, J. R. SubVirt: Implementing malware with virtual machines. In *Proceedings of IEEE Symposium on Security and Privacy* (2006), pp. 314–327.

[18] KOZUCH, M., AND SATYANARAYANAN, M. Internet suspend/resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications* (June 2002).

[19] MARUYAMA, H., SELIGER, F., NAGARATANAM, N., EBRINGER, T., MUNETOH, S., YOSHIHAMA, S., AND NAKAMURA, T. Trusted platform on demand (TPod). In *IBM Research Report* (2004).

[20] MCCUNE, J. M., PERRIG, A., AND REITER, M. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of USENIX Annual Technical Conference* (June 2006).

[21] NAOR, M., AND PINKAS, B. Visual authentication and identification. In *Advances in Cryptology — CRYPTO '97* (1997), B. S. Kaliski, Jr., Ed., vol. 1294 of *Lecture Notes in Computer Science*, International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, pp. 323–336.

[22] Next-Generation Secure Computing Base (NGSCB). `http://www.microsoft.com/resources/ngscb/default.mspx`, 2003.

[23] NICK L. PETRONI, J., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot, a coprocessor-based kernel runtime integrity monitor.

[24] OPREA, A., BALFANZ, D., DURFEE, G., AND SMETTERS, D. K. Securing a remote terminal application with a mobile trusted device. In *20th Annual Computer Security Applications Conference (ACSAC'04)* (2004).

[25] PFITZMANN, A., PFITZMANN, B., SCHUNTER, M., AND WAIDNER, M. Trusting mobile user devices and security modules. *IEEE Computer 30*, 2 (Feb. 1997), 61–68.

[26] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. Virtual network computing. *IEEE Internet Computing 2*, 1 (1998), 33–38.

[27] SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of USENIX Security Symposium* (2004), pp. 223–238.

[28] SATYANARANYANAN, M. KOZUCH, M.A., HELFRICH, C.J., O'HALLARON, D. Towards Seamless Mobility on Pervasive Hardware. *Pervasive and Mobile Computing 1*, 2 (2005), 157–189.

[29] SESHADRI, A., LUK, M., SHI, E., PERRIG, A., VAN DOORN, L., AND KHOSLA, P. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2005).

[30] Trusted Computing Group (TCG). `https://www.trustedcomputinggroup.org/`, 2003.

[31] Technical overview of terminal services. `http://www.microsoft.com/windowsserver2003/techinfo/overview/termserv.m%spx`, January 2005.

[32] TEIGLAND, D., AND MAUELSHAGEN, H. Volume managers in linux. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference* (June 2001).

[33] WANT, R., PERING, T., DANNEELS, G., KUMAR, M., SUNDAR, M., AND LIGHT, J. The personal server: Changing the way we think about ubiquitous computing. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing* (London, UK, 2002), Springer-Verlag, pp. 194–209.