Protecting Cryptographic Secrets and Processes

T. Paul Parker
Department of Computer Science, University of Texas at San Antonio

Abstract

Modern commodity operating systems are increasingly used to perform cryptographic operations such as digital signatures and to store cryptographic keys and data which the user considers private. Even when this data is encrypted, as is the recommended best practice, the encryption keys themselves are not well-protected. We examine the problem of protecting cryptographic secrets and cryptographic operations on commodity hardware and operating systems, with optional use of the TPM security hardware, and propose and evaluate some solutions. Our solutions are aimed primarily at attacks committed by malware and are generally applicable to maintaining the confidentiality and security of non-cryptographic secrets and processes.

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Dissertation Overview . . . 4
1.2.1 Safekeeping Cryptographic Keys from Memory Disclosure Attacks . . . 4
1.2.2 Protected Monitor . . . 5
1.2.3 Secure Signature Service Provider . . . 5
1.3 Combining the Pieces from the Chapters . . . 7

2 Safekeeping Cryptographic Keys from Memory Disclosure Attacks **8**
2.1 Introduction . . . 8
2.2 General Threat Model . . . 10
2.3 The Safekeeping Method and Its Implementation . . . 11
2.3.1 Basic Idea and Resulting Prototype . . . 12
2.3.2 Scrambling and Dispersing a Key in RAM . . . 12
2.3.3 Obscuring the Index Table . . . 14
2.3.4 Disabling Interrupts . . . 16
2.4 Refining Attacks By Considering Our Design . . . 17
2.5 Security Analysis . . . 18
2.5.1 Example Scenario . . . 19
2.5.2 Effects of the Key Compromise Methods . . . 20
2.5.3 Security Summary . . . 23
2.6 Performance Analysis of Prototype . . . 24
2.7 Conclusion and Open Problems . . . 25

3 Securing Digital Signing with the Protected Monitor **27**
3.1 Introduction . . . 27
3.2 Assured Digital Signing . . . 28
3.2.1 System Logical Design . . . 30
3.2.2 System Design . . . 31
3.2.3 Implementation . . . 41
3.3 Security Analysis . . . 43
3.3.1 Defeating Attacks against Domain U Components . . . 44
3.3.2 Defeating Attacks against Domain 0 Components . . . 47
3.3.3 Defeating Non-Domain-Specific Attacks . . . 47
3.4 Experimental Evaluation of Performance . . . 49
3.4.1 Microbenchmark Performance of Inter-VM Communication . . . 49
3.4.2 Assured Signing Performance . . . 50
3.5 Conclusion . . . 51

4 Related Work **52**
4.1 Introduction . . . 52
4.2 Protecting Secrets with Special Hardware . . . 53
4.2.1 Hardware Solutions: Trusted Platform Module . . . 53
4.2.2 Protecting Functions with TPM . . . 55
4.2.3 Other Hardware Solutions . . . 55
4.3 Protecting Secrets and Functions with Virtual Machines . . . 56
4.3.1 Sujit Sanjeev Master's Thesis . . . 58
4.3.2 Other Virtual Machine Related Work . . . 58
4.4 Protecting Secrets via Conventional Software . . . 59
4.4.1 Protecting Keys . . . 59
4.4.2 Microsoft Windows Key Protection . . . 60
4.4.3 Protecting General Secrets . . . 60
4.5 Protecting Keys Cryptographically . . . 61
4.6 Work Specifically Related to Securing Signatures . . . 61
4.7 Work Specifically Related to the Protected Monitor . . . 62

5 Conclusion **64**
5.1 Summary . . . 64
5.2 Future Work . . . 65

Bibliography **66**

A List of Author's Publications and Presentations **73**
A.1 Security Publications . . . 73
A.2 Security Presentations . . . 74
A.3 Previous Publications . . . 74

Chapter 1

Introduction

1.1 Motivation

Computer security has made impressive theoretical gains, such as the proofs of security for cryptographic protocols. However, the security of systems in practice depends on the security of cryptography in practice. The security of cryptography in practice depends on the security of the entire system, particularly the security of the cryptographic keys and cryptographic processes in real systems, and unfortunately this area has not been well-studied. Worse, ordinary users of commodity hardware and operating system software frequently find themselves besieged by spam, malware, and other security-related issues which can contribute to reduced system security and further exacerbate the reduction of security of cryptography in practice.

In this dissertation we emphasize securing cryptographic secrets (keys) and cryptographic processes (especially digital signatures) while still allowing the user to run ordinary hardware and software. In particular, we generally allow the user to run existing applications and operating systems completely unmodified.

We note that cryptographic keys are a particularly important type of critical secret. There are two consequences to this fact: One, much of computer security is dependent on the confidentiality of cryptographic keys. Two, techniques that are applied for preserving the confidentiality of critical secrets and other confidential data can often be applied to securing cryptographic keys. The converse is sometimes true: techniques used to secure cryptographic keys can sometimes be used to secure other types of critical secrets.

Most of this dissertation will focus on securing cryptographic keys and cryptographic operations, although our design and much of our implementation can be used to secure other kinds of critical secrets and processes.

We generally seek to secure cryptography against malware, although certain of our solutions are more general. Using malware to disclose keys and other critical secrets is not difficult.

The overall goal of this work may then be summarized thus:

To protect critical secrets and processes, especially cryptographic keys and digital signatures, from software attacks, particularly attacks by malware.

1.2 Dissertation Overview

This dissertation presents three significant mechanisms, each of which is realized by a software component. These are shown in Figure 1.1. We will explain these mechanisms in this order, which is the same as the chapter order, and then in the next section will see how this represents a dependency ordering in a possible integration of the components from the chapters.

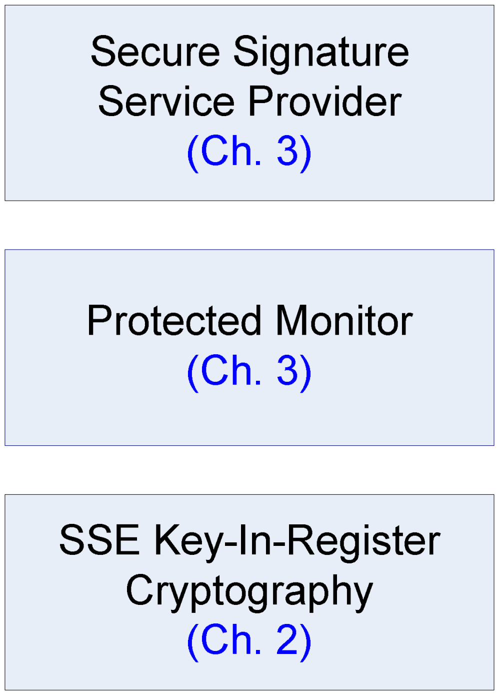

Secure Signature Service Provider
(Ch. 3)
Protected Monitor
(Ch. 3)
SSE Key-In-Register Cryptography
(Ch. 2)

Figure 1.1: Overview of Components. Chapter numbers are set in parentheses.

1.2.1 Safekeeping Cryptographic Keys from Memory Disclosure Attacks

Chapter 2 presents and analyzes a technique for using a cryptographic key without ever having the key in memory. This gives protection against memory disclosure attacks which otherwise can frequently recover keys, particularly in the case of Apache on Linux [32]. As a specific example, a prototype is created that modifies RSA private key encryption in OpenSSL to use the technique. The contributions include:

1. No special hardware (e.g., TPM) is required; only resources found in typical CPU's are used.
2. The scheme is shown to leave no words of the private key exponent d in RAM.
3. A RAM scrambling technique, which must be used to store the key in the single-CPU-core case, is evaluated, showing that common attacks such as entropy scanning, signature scanning, and content scanning are infeasible.

1.2.2 Protected Monitor

Chapter 3 includes a foundational piece which we believe may be of independent value. This foundation piece, called the *protected monitor* provides a platform on which secured services can be built. It is particularly well-suited to securing against malware attacks, although it can be used for many other types of attacks. The monitor's architecture, depicted in Figure 1.2, relies on a virtual machine manager to provide memory protection but still allows the monitor to operate from within the memory space of the virtual machine, unlike virtual machine introspection. Further details will be explained in Chapter 3; in the meantime we summarize the contributions here:

1. No hardware support is required.
2. Monitor has complete memory protection from user VM.
3. Services built on the platform can optionally interact with the kernel even though the platform is not dependent on kernel integrity.
4. The Protected Monitor is secure against almost all attacks from the user VM, including kernel compromise (e.g., rootkits) and userland compromise.

1.2.3 Secure Signature Service Provider

Chapter 3 presents the Secure Signature Service Provider. This secures both the keys used for signing and the signing process itself, even in the presence of malware running at elevated privilege levels.

Figure 1.3 depicts the architecture of this system. Further details will be explained in Chapter 3; in the meantime we summarize the contributions here:

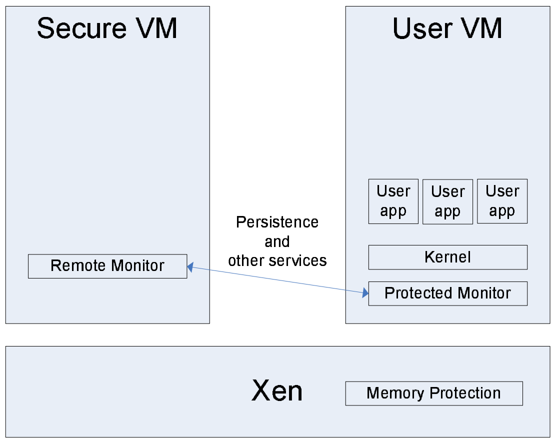


Figure 1.2: Architecture of Protected Monitor (Chapter 3)

1. No hardware support is required, although a TPM can be used for additional remote verifiability of signatures.
2. Signature requests are validated using four criteria: (i) static measurement of boot and kernel (using TPM); (ii) secured crypto library; (iii) authentication of the requesting program (measure binary); (iv) trusted path user confirmation dialog.
3. Key storage services are secure against malware and even raw disk access (from within the VM).
4. Signature request processing is likewise secure against malware.
5. The design provides for a smaller TCB for signature processing operations, since the cryptography implementation can rely on a smaller and controlled software stack.

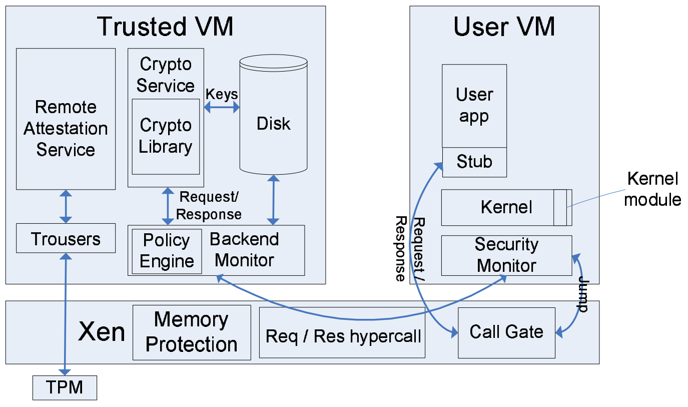


Figure 1.3: Architecture of Secure Signature Service Provider (Chapter 3)

Location	*Keys*
VM Disk Protection	Signature Service Provider (Ch. 3)
VM RAM Protection	Signature Service Provider (Ch. 3)
Physical RAM Protection	SSE Key-In-Register Cryptography (Ch. 2)

Table 1.1: Type of protection offered by different pieces. Parentheses contain chapter numbers.

1.3 Combining the Pieces from the Chapters

The reader may wish to understand how the various pieces we propose fit together. One way to understand this is to examine the function of the protection pieces (Chapters 2 and 3).

The table in Figure 1.1 shows this:

- The entire system is built on top of the protected monitor (Chapter 3).
- SSE Key-in-Register Cryptography (Chapter 2) protects the key from any RAM attack, including disclosure of physical RAM (e.g., via a Firewire attack).
- The signature service provider (Chapter 3) protects keys on the VM's disk as well as in the VM's RAM.

Another way to understand this is to see how they could be used together. Figure 1.4 depicts a possible system architecture that uses all of the pieces proposed in this dissertation. Keys are never left in memory, but are used directly out of registers (Chapter 2). The entire system is built on the protected monitor (Chapter 3). The Secure Signature Service Provider (Chapter 3) uses the key-in-register cryptography for its cryptographic operations, and provides digital signature services to cryptographic applications as well.

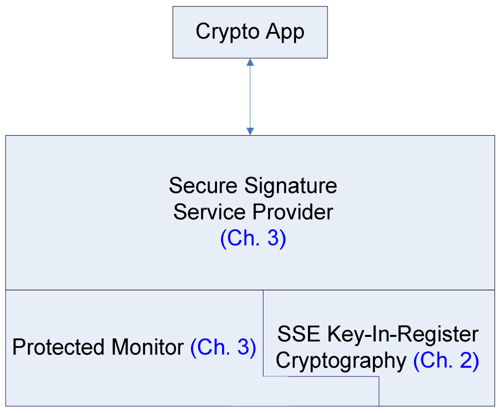

Crypto App
Secure Signature Service Provider (Ch. 3)
Protected Monitor (Ch. 3)
SSE Key-In-Register Cryptography (Ch. 2)

Figure 1.4: An Example Architecture Combining All Chapters. Chapter numbers are set in parentheses.

Chapter 2

Safekeeping Cryptographic Keys from Memory Disclosure Attacks

2.1 Introduction

How should we ensure the secrecy of cryptographic keys during their use in RAM? This problem is important because it would be relatively easy for an attacker to have unauthorized access to (a portion of) RAM so as to compromise the cryptographic keys (in their entirety) appearing in it. Two example attacks that have been successfully experimented with are those based on the exploitation of certain software vulnerabilities [32], and those based on the exploitation of Direct Memory Access (DMA) devices [53]. In particular, [32] showed that, in the Linux OS versions they experimented with, a cryptographic key was somewhat flooding RAM, meaning that many copies of a key may appear in both allocated and unallocated memory. This meant an attacker may only need to disclose a small portion of RAM to obtain a key. As a first step, they showed how to ensure only one copy of a key appears in RAM. Their defense is not entirely satisfactory because the success probability of a memory disclosure attack is then roughly proportional to the amount of the disclosed memory. Their study naturally raised the following question: *Is it possible, and if so, practical, to safekeep cryptographic keys from memory disclosure attacks without relying on special hardware devices?* The question is relevant because legacy computers may not have or support such devices, and is interesting on its own if we want to know what is feasible without special hardware devices. (We note that the basic idea presented in this chapter may also be applicable to protect cryptographic keys appearing in the RAM of special hardware devices when, for example, the devices' operating systems have software vulnerabilities that can cause the disclosure of RAM content.)

Our contributions. In this chapter we affirmatively answer the above question by making three contributions. First, we propose a method for exploiting certain architectural features (i.e., certain CPU registers) to safekeep cryptographic keys from memory disclosure attacks (i.e., ensure a key never appears in its entirety in the RAM). Nevertheless, cryptographic functions are still efficiently computed by ensuring that a cryptographic key appears in its entirety in the registers. This may sound counter-intuitive at first glance, but is actually achievable as long as the registers can assemble the key on-the-fly as needed.

Second, as a proof of concept, we present a concrete realization of the above method based on OpenSSL, by exploiting the Streaming SIMD Extension (SSE) XMM registers of modern Intel and AMD x86-compatible CPU's [21]. The registers were introduced for multimedia application purposes in 1999, years before TPM-enabled computers were manufactured (TCG itself was formed in 2003 [30]). Specifically, we conduct experimental studies with the RSA cryptosystem in the contexts of SSL 3.0 and TLS 1.0 and 1.1. Experimental results show that no portion of a key appears in the physical RAM (i.e., no portion of a key is spilled from the registers to the RAM). The realization is not straightforward, and we managed to overcome two subtle problems:

1. Dealing with interrupts: For a process that does not have exclusive access to a CPU core (i.e., a single-core CPU or a single core of a multi-core CPU), we must prevent other processes from reading the SSE XMM registers. This requires us to prevent other processes from reading the registers by disabling interrupts, and to avoid entering the kernel while the key is in the registers (this is fortunately not difficult in our case). Because applications such as Apache generally do not run with the `root` privilege that is required for disabling interrupts, we designed a Loadable Kernel Module (LKM) to handle interrupt-disabling requests issued by applications such as Apache.

2. Scrambling and dispersing a cryptographic key in RAM while allowing efficient re-assembling in registers: Some method is needed to load a cryptographic key into the registers in a secure fashion; otherwise, a key may still appear in RAM. For this, we implemented a heuristic method for "scrambling" a cryptographic key in RAM and then "re-assembling" it in the relevant registers.

Third, we articulate an (informal) adversarial model of memory disclosure attacks against cryptographic keys in software environments that may be vulnerable. The model serves as a systematic basis for (heuristically) analyzing the security of software against memory disclosure attacks, and may be of independent value.

Discussion on the real-world significance. As will be shown in the case study prototype system, the method proposed in this chapter can be applied to legacy computers that have some architectural features (e.g., x86 XMM registers or other similar ones). Two advantages of a solution based on the method are

(1) it can be obtained for free, and (2) it could be made transparent to the end users; both of these ease real-world adoption. However, we do not expect that the solution will be utilized in servers for processing high-throughput transactions, in which case special high-speed and high-bandwidth hardware devices may be used instead so as to accelerate cryptographic processing. Nevertheless, our solution is capable of serving 50 new HTTPS connections per second in our experiments. The attacks addressed in this chapter are memory disclosure attacks, which are mainly launched via the exploitation of software vulnerabilities in operating systems. Dealing with attacks against the application programs is beyond the scope of the present work.

Chapter outline. The rest of this chapter is organized as follows. Due to the complexity of the adversarial model, we specify attacks against based on two dimensions. One dimension is *independent* of our specific solution and is elaborated in Section 2.2 because it guides the design of our specific solution. The other dimension is *dependent* upon our solution (e.g., the attacker may attempt to identify weaknesses specific to our solution) and presented in Section 2.4, after we present our specific solution in Section 2.3. Section 2.5 informally analyzes the security of the resulting system. Section 2.6 reports the performance of our prototype. Section 2.7 concludes the chapter with some open problems. Note that related work is discussion in Chapter 4.

2.2 General Threat Model

Independent of our specific solution design, we consider a *polynomial-time* attacker who can disclose some portion of RAM through some means that may also give the attacker some extra power (as we discuss below). To make this concrete, in what follows we present a classification of the most relevant memory disclosure attacks (see also Figure 2.1).

Pure memory disclosure attacks. Such attackers are only given the content of the disclosed RAM. Depending on the amount of disclosed memory, these attacks are divided into two cases: *partial* memory disclosure and *full* memory disclosure. Furthermore, partial disclosure attacks can be divided into two cases: *untargeted* partial disclosures and *targeted* partial disclosures. An untargeted partial attack discloses a portion of memory but does not allow the attacker to specify which portion of the memory (e.g., random portions of RAM that may or may not have a key in it). In contrast, a targeted partial attacker somehow allows the attacker to obtain a specific portion of RAM. Although we do not know how to accomplish this, this may be possible for some sophisticated attackers.

Augmented full memory disclosure attacks. Compared with the full memory disclosure attacks where attackers just analyze the byte-by-byte RAM content, augmented full memory disclosures give the attacker

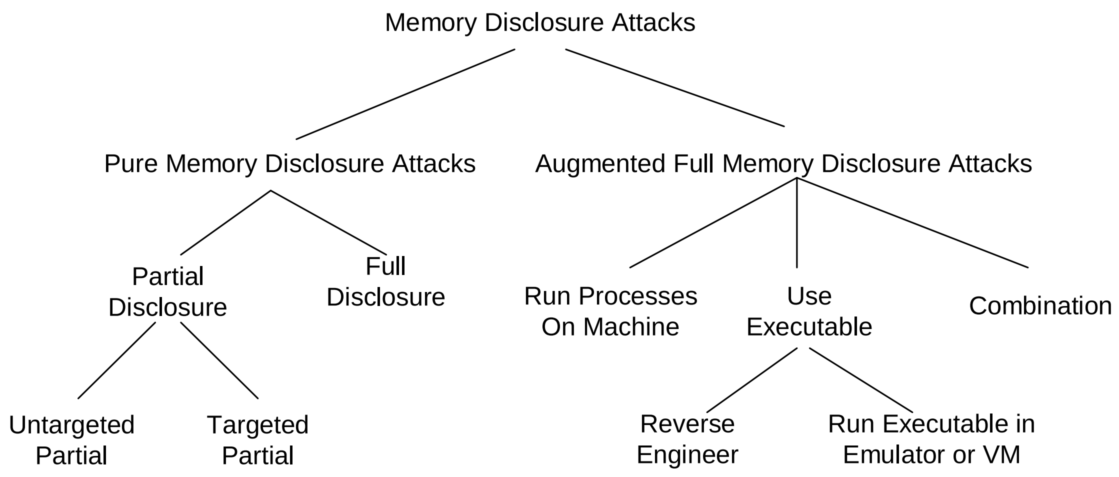


Figure 2.1: Memory disclosure attack taxonomy.

extra power. The first possible augmentation is to allow the attacker to run processes on the machine that is being attacked. This requires the attacker to have access to a user account on the machine, but neither `root` nor the account that owns the key being protected (e.g., `apache`); otherwise, we cannot hope to defeat the attacker. The main trick here is that the attacker here may seek to circumvent the ownership of the registers that store the key (if applicable). The second possible augmentation is for the attacker to use the victim user's own executable image (which is probably in the disclosed RAM) to recover the key, which is possible because the executable together with its state must be able to recover the key. We further classify this augmentation into two cases: *reverse-engineering*, where the attacker reverse-engineers the executable and state to recover the key; and *running the executable in an emulator or VMM*, where the attacker can actually execute the entire disclosed memory image and discover (for example) what is put in the disclosed RAM or registers, if the attacker can somehow simulate the unknown non-RAM state such as CPU registers. Finally, an attacker could employ multiple augmentations simultaneously, which we we label as "combination" in our classification.

2.3 The Safekeeping Method and Its Implementation

In this section we first discuss the basic idea underlying our method, and then elaborate the relevant countermeasures that we employ to deal with threats mentioned above (this explains why we said that the threat model guided our design).

2.3.1 Basic Idea and Resulting Prototype

The basic idea of our method is to exploit some modern CPU architectural features, namely large sets of CPU registers that are not heavily used in normal computations. Intuitively, such registers can help "avoid" cryptographic keys appearing in RAM during their use, because we can make a cryptographic key appear in RAM only in some scrambled form, while appearing in these registers in cleartext and in its entirety. In our prototype, we use the x86 XMM register set of the SSE multimedia extensions, which were originally introduced by Intel for floating-point SIMD use and later also adopted by AMD. Each XMM register is 128 bits in size. Eight such registers, totaling 1024 bits, are available in 32-bit architectures; 64-bit architectures have 16, for a total of 2048 bits. These registers can be exploited to run cryptographic algorithms because a 32-bit x86 CPU can thus store a 1024-bit RSA private exponent, and a 64-bit one can store a 2048-bit exponent.

Our prototype is based on OpenSSL 0.9.8e, the Ubuntu 6.06 Linux distribution with a 2.6.15 kernel, and SSE2 which was first offered in Intel's Pentium 4 and in AMD's Opteron and Athlon-64 processors. Figure 2.2 depicts the resulting system architecture. It adds a new *supporting mechanism* layer that loads

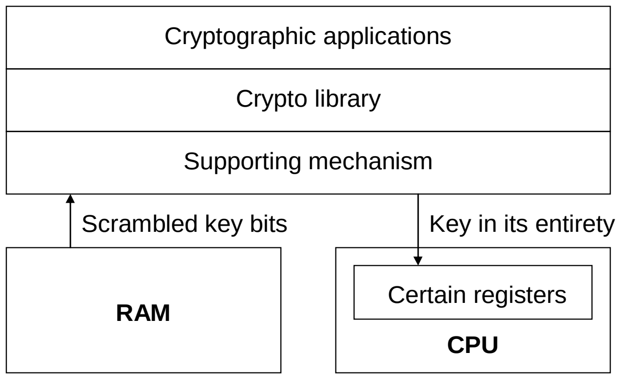


Figure 2.2: The resulting system architecture

a scrambled key into the relevant registers (i.e., assembling the scrambled key into the original key) and makes it available to cryptographic routines.

2.3.2 Scrambling and Dispersing a Key in RAM

A crucial issue in our solution is to store the key in RAM such that it will be difficult for attackers to compromise. For this, one may suggest to encrypt the key in RAM and then decrypt and put the key directly into registers.

However, this approach has two issues that are not clear: (i) where the key for this "outer" layer of encryption can be safely kept (i.e., we now have a *chicken-and-egg* problem, because that key needs to be encrypted too), and (ii) how to ensure that there is no intermediate version of the key in RAM. A similar argument would also be applicable to other techniques aimed for a similar purpose. As such, we adopt the following heuristic method for scrambling and dispersing a key in RAM:

- Initialization: This operation prepares a dispersed scrambled version of the key in question such that the resulting bit strings are stored on some secure storage device (e.g., harddisk or memory stick) and thus can later be loaded into RAM as-is. This can be done in a secure environment and the resulting scrambled key may be kept on a secure storage device such as a memory stick.

- Recovery: the key in its scrambled form is first loaded into RAM, and then somehow "re-assembled" at the relevant registers so that the key appears in its entirety in the registers.

As illustrated in Figure 2.3, the *initialization* method we implemented proceeds as follows. (i) The

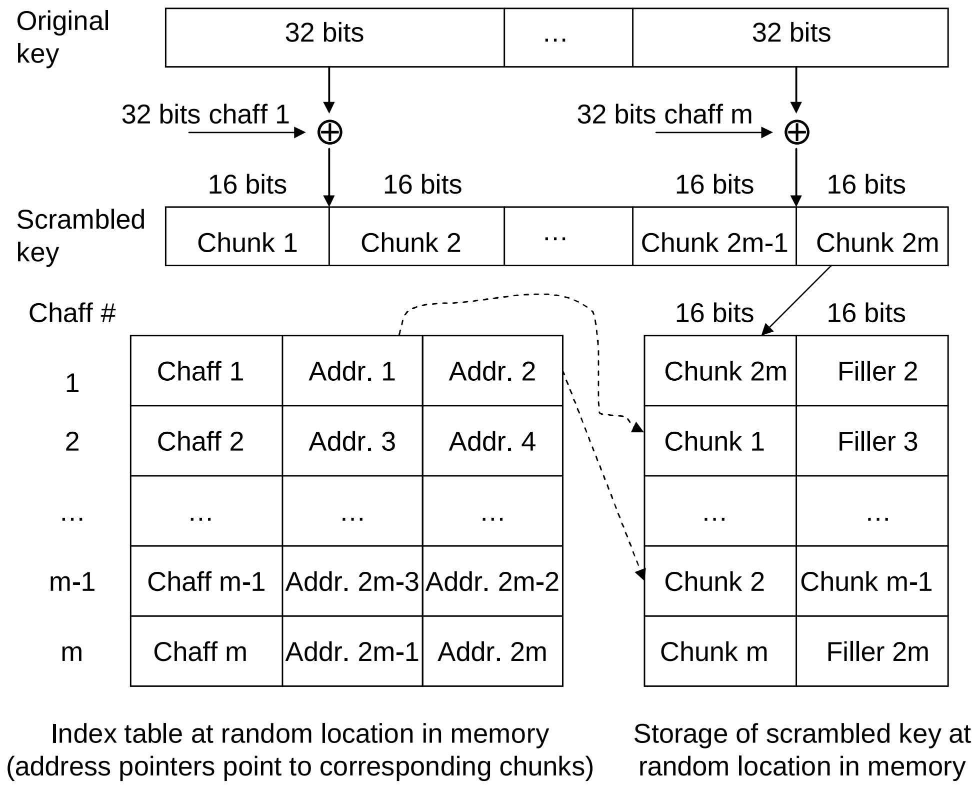


Figure 2.3: Prototype's method for scrambling and dispersing key

original key is split into blocks of 32 bits. Note that the choice of 32-bit words is not fundamental to the design, it could be a 16-bit word or even a single byte. (ii) Each chunk is XORed with a 32-bit chaff that is independently chosen. As a line of defense, it is ideal that the chaffs do not help the attacker to identify the whereabouts of the index table. (iii) Each transformed block is split into two chunks of 16 bits. (iv) The chunks are mixed with some "fillers" (i.e., useless place-holders to help hide the chunks) that exhibit similar characteristics as the chunks (e.g., entropy-wise they are similar so that even the entropy-based search method [57] cannot tell the fillers and the chunks apart). Clearly, the *recovery* can obtain the original key according to the index table, each row of which consists of a chaff and the address pointers to the corresponding chunks. Since security of the index table is crucial, in the next section we discuss how to make it difficult to compromise.

We note that some form of All-Or-Nothing-Transformation [14] (as long as the inversion process can be safely implemented in the very limited environment of registers) should be employed prior to the scrambling in order to safeguard against attacks that work on portions of RSA keys (e.g., [10] gives an attack that can recover an RSA private key in polynomial time given the least-significant $n/4$ bits of the key). Using such a transformation protects our scheme from these attacks and insulates the scheme and analysis from progress in partial-exposure key breaking work. This also protects our scheme from attacks that exploit structure in the RSA key, such as some attacks from Shamir and van Someren [57]. The exact technique and implementation should be be chosen carefully so as to not spill any intermediate results into RAM.

2.3.3 Obscuring the Index Table

To defend against an attacker who attempts to find and follow the sequence of pointers to the index table, we can adopt the following two defenses.

First defense. We can use a randomly-chosen offset for all the pointers in the table, as well as a randomly-chosen delta number to modify the data values themselves. The offset and delta are chosen once before the table is constructed, and then the pointer values in the table are actually the memory location minus the offset. The actual data values stored at the memory locations are the portions of the key minus the delta value. This means that even if the attacker finds the table, the pointers in it are not useful without successfully guessing the offset and delta.

We must prevent the attacker from simply scanning all of the statically-allocated data for potential offset and delta values and trying all of them whenever interpreting a possible table pointer. We can defend against this by using (for example) 16 numbers as the set of potential pointer offsets, and 8 numbers as the set of potential delta values. A random number chosen at compile-time determines whether the actual pointer or value is or is not XOR'd with each member of the corresponding set. (`make` can compile and run a short program to generate this number and emit it as a `#define` suffixed to a header file. Such values do not have storage allocated and only appear in the executable where they are used.) Carefully constructing an expression controlled by this value but where the appearance of the value itself can be optimized away by the compiler means compiler optimization techniques will ensure that this constant does not appear directly in the final executable (and therefore cannot be read from a RAM dump). [1] We will show an example expression below, using a conceptual syntax for clarity.

Each number in the set is the same size as the pointer or short value. At compile time one bit determines whether to XOR the two high halves, and the following bit whether to XOR the two low halves. Note that breaking each number into two separately-operated pieces is useful because it squares the factor that we are increasing the attacker's search space by. The use of each set forces the attacker to examine 4^{16} and 4^{8} possibilities for the pointers and short values, respectively. Let us refer to the 64-bit set of numbers as $64B_0..64B_{15}$, and designate the top and bottom halves of these as $64B_0^T..64B_{15}^T$ and $64B_0^B..64B_{15}^B$ respectively, and use p to denote the pointer being masked. Then,

$$p = p \oplus (64B_0^T \wedge bit_0) \oplus (64B_0^B \wedge bit_1) ... \oplus (64B_{15}^T \wedge bit_{30}) \oplus (64B_{15}^B \wedge bit_{31})$$

where $\wedge$ is an operator that returns 0 if either operand is zero, and returns the first operand otherwise. The computation is similar for the 16-bit short values that contain scrambled RSA key pieces.

Second defense. Let us suppose the attacker has some magical targeted partial disclosure attack that identifies the index table, chunks, offset XOR values, and delta XOR values (note the actual possible attacks we know of are not nearly this powerful). The control values for the offset XOR can be efficiently computed using the chunk addresses, and the control values for the delta XOR may then be computed with a cost of 2^{16}.

In order to rigorously defend against this, we can add a compile-time constant (see Section 2.3.3) that is used to specify a permutation on the index table. Lookups on the index table will now use this constant to control the order (e.g., the index used would be the index sought plus the last several bits ($\lg t$, t is table size) of a pseudo-random number generator based on the pointer, modulus t. The pseudo-random number generator must have small state (current value kept in a register), be possible to compute entirely inside the x86 register space (limiting on 32-bit but roomy for 64-bit), and the trailing bits must not repeat within a period t). A 32-bit permutation constant (seed) would increase the attacker's search space by a factor of 2^{32}; a larger constant could be used if that simplified the implementation while providing at least 2^{32} permutations.

Discussion. Without these defenses, an attacker could just build the executable on an identical system, run `objdump` and look for the appropriate variable name, and then examine that memory location in the

[1] We verified a sample expression compiled to a sequence of appropriate XOR's, with the random constant not appearing, in gcc 3.4 and 4.0, with -O2.

process to find the index table (this omits some details such as how to recover the process page table which gives the virtual memory mapping). With these defenses, the attacker must locate and interpret particular sequences of assembly language instructions in the particular executable being used on this machine to determine how to unscramble and order pointers and values in each of various stages in the scrambling process. The possible attack routes are explained in Section 2.4 and analyzed in Section 2.5.

2.3.4 Disabling Interrupts

In order to ensure that register contents are never spilled to memory (for a context switch or system event), we need to disable interrupts. This can be achieved by disabling interrupts via, for example, a kernel module that provides a facility for non-`root` processes to disable and enable interrupts on a CPU core. However, there are three important issues:

1. Since illegitimate processes could use the interrupt-disabling functionality to degrade functionality or perform a denial-of-service attack, care must be taken as to which programs are allowed to use this facility. A mechanism may be used to harden the security by authenticating the application binary that requests disabling interrupts, e.g., by verifying a digital signature of the binary.

2. The interrupt-disabling facility itself may be attacked. For example, the kernel module we use to disable interrupts could be compromised or faked so that it silently fails to disable interrupts. Fortunately, we can detect this omission from userland and refuse to populate the XMM registers, reducing the attacker to a denial-of-service attack, which was already possible because the attacker had to have kernel access.

3. A clever attacker might be able to prevent the kernel module from successfully disabling interrupts. For example, the attacker might perpetrate a denial-of-service attack on the device file used to send commands to the kernel module. Two points of our design make this particular attack difficult for the attacker:

   (a) First, the kernel module allows multiple processes to open the device file simultaneously, so that multiple server processes can access it, meaning an attacker cannot open the device to block other processes.

   (b) Second, the code that calls the kernel module automatically retries if interrupts have not become disabled. So in the worst case, the attack is downgraded to a denial-of-service attack, which is already easy when the attacker has this level of machine access.

Discussion. Disabling interrupts could cause side-effects, most notably with real-time video, or dropping network traffic if interrupts were disabled for a long time, which would cause a retransmission and hence some bandwidth and performance cost. Having multiple cores, as most 64-bit machines and almost all new machines do, would mitigate these problems.[2] Moreover, no ill effects were observed from disabling interrupts on our systems. Note that non-maskable interrupts such as page faults and system management interrupts cannot be disabled on x86. Thus the scheme is susceptible to low-level attacks that modify their handlers. Such attacks require considerable knowledge and skill, require privileges on well-managed systems, and are frequently hardware-specific; we do not deal with such attacks in the present work.

2.4 Refining Attacks By Considering Our Design

Now we consider what key compromise methods may be effective against our design. We emphasize these attacks include methods specific to our solution and thus are distinct from the general threat model, whose classes of attacks are independent of our solution and regulate the resources available to the attacker. These methods specify the rows of our attack analysis chart (Figure 2.4), whereas our threat model specifies the columns. The short designation used in the figure to name these parts is highlighted for easy reference when examining the figure. Often multiple approaches can be used to achieve the same goal, so sometimes the attack chart lists two ways to accomplish a goal, with an `OR` after the first. When multiple steps are needed to accomplish a goal, they are individually numbered. Here we list and explain the methods found in the table:

- **Retrieve key from registers**. The attacker may attempt to compromise the key by reading it directly from the XMM registers.
- **Retrieve key directly from RAM**. The attacker may try to read the key directly from RAM, if present.
- **Descramble key from RAM**. These are the most interesting and subtle attack scenarios. Again, since multiple approaches may be used to achieve the same attack effect, sometimes the attack chart lists two ways to accomplish a given objective, with an `OR` after the first (see Figure 2.4). Moreover, when multiple steps are needed to accomplish an objective, they are individually numbered. The descrambling attacks may succeed via two means: index table or chunks.

[2]In fact, according to `/proc/interrupts`, the Linux 2.6.15 kernel we used directed all external interrupts to the same core, so simply using the other cores for our technique would avoid the problem entirely.

– *Via index table*. This attack can be launched in three steps (see also Figure 2.4): "1.Locate index table", "2.Interpret index table", and "3.Follow pointers". Specifically, the attacker must first locate the table by scanning RAM for it (e.g., using an entropy scan) or by following pointers to it. Assuming the attacker successfully locates the table, the attacker must then determine how to properly interpret it, since the pointers are scrambled and the chunk chaff values are scrambled also (per Section 2.3.3). One way to interpret the table is to somehow compute the actual XOR used on the offsets and compute the actual XOR used on the values, "Determine actual XOR offset and XOR delta". Another way is to "Use deltas and offsets and determine combination", this means to find the deltas and offsets and then determine the proper combination of them (i.e., the value of the control variable embedded in the executable specifying whether to use each individual delta and offset). Finally, if the attacker has successfully located the table and determined how to interpret the table itself, the pointers must be followed to actually find the chunks in proper order. In Section 2.3.3 we discussed how to defend against this by introducing a substantial number of permutations.

– *Via chunks*. The attacker can avoid interpreting the table and attempt to work from the chunks directly. This requires three steps (see also Figure 2.4). First, the attacker must locate the chunks themselves in the memory dump ("1.Locate chunks"). Then, the attacker must interpret the chunks ("2.Interpret chunks") that were XOR'd with the chaff values. Lastly, the attacker must determine the proper order for the chunks ("3.Order chunks"), which is demanding since the number of permutations is considerable.

2.5 Security Analysis

It would be ideal if we could rigorously prove the security of the resulting system. Unfortunately, this is challenging because it is not clear how to formalize a proper theoretic model. The well-articulated models, such as the ones due to Barak et al. [6] and Goldreich-Ostrovsky [28], do not appear to be applicable to our system setting. Moreover, the aforementioned "supporting mechanism" itself may be reverse-engineered by the attacker, who may then recover the original key. We leave devising a formal model for rigorously reasoning about security in our setting as an open problem. In what follows we heuristically discuss security of the resulting system.

Figure 2.4 summarizes attacks against the resulting system, where each row corresponds to a key-compromise attack method (see Section 2.4) whereas the columns are the various threat models. At the intersection of a column and row is an attack effect, which is a one or two letter code that explains the

Key Compromise Method	Full Disclosure	Partial Disclosure Untargeted	Partial Disclosure Targeted	Reverse Engineer Executable	Run Executable in Emulator	Run Processes on Machine	Combination
Retrieve key from registers	n/a	n/a	n/a	n/a	**E (manual)**	A	**E (manual)**
Retrieve key directly from RAM	B	B	B	B	B	B	B
Descramble Key							
Via index table							
1. Locate index table							
Scan **OR**	C	C	S	C	n/a	C	C
Follow pointers	**DS**	**DS**	**S**	**S (manual)**	E (manual)	**DS**	E (manual)
2. Interpret index table							
Determine actual XOR offset and XOR delta **OR**	**F1**	**F1**	**S (if possible)**	**S (manual)**	E (manual)	**F1**	S (manual)
Use deltas and offsets and determine combination							
I. Find deltas and offsets AND	DD	DD	S (if possible)	S (manual)	E (manual)	DD	S (if possible)
II. Determine combination of each	F2	F2	F2	F2	F2	F2	F2
3. Follow pointers	**G**	**G**	**G**	**S (manual)**	E (manual)	**G**	S (manual)
Via chunks							
1. Locate chunks	DD	DD	S (if possible)	S (manual)	E (manual)	DD	E (manual)
2. Interpret chunks	H	H	H	H	E (manual)	H	E (manual)
3. Order chunks	I	I	I	I	E (manual)	I	E (manual)
Computational cost of best attack:	2^58	2^58	2^32 (if PDT possible)	1 (very manual)	1 (very manual)	2^58	1 (very manual)

Figure 2.4: Effects of different attack methods in different threat models. Legend: A — Retrieving key from registers fails. B — Retrieving key from RAM fails because no copy is there. C — Table scan fails because no identifying information. DD — Doable with caveat (dispersed). DS — Doable with caveats (no symbols). E — Run executable in emulator or virtual machine. F1 — Search 2^{26} possibilities for actual XOR offset and actual XOR delta. F2 — Search 2^{36} to determine XOR offset control value and XOR delta control value. G — Circumventing table compile-time constant ordering defense requires 2^{32}. H — Chunks encoded with 16 bits of chaff (per chunk). I — Chunks have 2^{296} possible orders. S — Attack stage would succeed given the caveat in parentheses. Bold items indicate best key compromise method in a given threat type. Notes in parentheses indicate caveats: "Manual" means requires substantial manual work for a highly-knowledgeable and skilled attacker, "if possible" means if there is a targeted partial disclosure attack that somehow finds only the items of interest.

2.5.1 Example Scenario

To aid understanding of the chart, we consider as an example the Full Disclosure threat model where the attacker is given the full RAM content and attempts to compromise the key in it. In this case, the specific attack "retrieving the key from registers" does not apply because RAM disclosure attacks do not contain the contents of registers. Moreover, the specific attack "retrieving the key from RAM" fails because RAM does not contain the key, as detailed in effect "B" in Section 2.5.2. Thus, the attacker may then try to retrieve the key via the index table, or via the chunks directly as elaborated below.

Via index table. Continuing down the column of the Full Disclosure threat model, the attacker scans the RAM dump for the index table, but this fails because the table has no readily-obvious identifying information (code "C" in Figure 2.4). Instead, the attacker can build the executable on another machine so as to find the storage location for the pointer to the index table, as shown in code "DS" in Figure 2.4. The attacker may try to guess the actual XOR value used for pointer offsets and the actual XOR value used for chunk deltas ("F1" in Figure 2.4), but the search space is 2^{26}, which will still have to be multiplied by later cost factors since the guess can't be verified until the actual key is assembled. Instead, the attacker can find the values that are combined to produce the deltas (difficult because they are dispersed throughout the process memory "DD"), and then determine what combinations of these are used to form the actual offset XOR value and the actual delta XOR value ("F2"), at a cost of 2^{36} different guesses. In order to actually follow the decoded pointers and reassemble the keys, the 2^{32} permutation induced by a compile-time random value ("G") must be reversed, which requires considering 2^{32} permutations for each of those 2^{36} guesses. Thus $2^{32} \cdot 2^{36} = 2^{68}$ keys must be examined to attack via the index table if the deltas and offsets are found and then their combinations examined. Since directly determining the offsets and deltas costs 2^{26} ("F1"), examining 2^{32} permutations for each of those yields a cheaper total cost of 2^{58}. As we will see, this is the most efficient attack, so "DS" "F1" and "G" are bolded because together they form the best attack for this column.

Via chunks. In this case the chunks must first be located from dispersed memory, with no particular identifying characteristics ("DD"). The chunks must then be decoded, which is difficult since each has been XOR'd with its own random 16-bit quantity ("H") which is stored only in the index table (breaking this is prohibitively expensive because individual chunks can't be verified, e.g., a 1024-bit key has 64 16-bit chunks, so ${2^{16}}^{64} = 2^{1024}$). Lastly, the chunks must be ordered, but there are 2^{296} possible orders ("I"), so clearly the index table attack above that yields 2^{58} possible keys is faster.

Computational Cost of Best Attack. The fastest attack for the Full Disclosure threat model was the index table attack that yields 2^{58} possible keys. $2^{58} = 2.9 * 10^{17}$, meaning an adversary with 8 cores that can each check 1000 RSA keys per second (i.e., 1000 sign operations per second per core) could break the defense to recover the key in slightly more than a million years (about ten million CPU years).

2.5.2 Effects of the Key Compromise Methods

Here we elaborate the effects of the key compromise methods in the threat models. For example, effect **A** is what occurs when an attacker launches the attack "retrieve the key from registers" in the threat model of “run processes on machine”.

Effect A: Retrieving key from registers fails. The most obvious key compromise method is to steal the key when it is loaded into the SSE registers. As discussed before, special care was taken to prevent this attack by appropriately disabling interrupts, so that our process has full control of the CPU until we relinquish it.

Effect B: Retrieving key from RAM fails because no copy is there. The second most obvious way to recover the key is if it was somehow “spilled” from the registers to RAM during execution. We conducted experiments to confirm that this does not happen. Specifically, we analyzed RAM contents while Apache is running under VMware Server on an Intel Pentium 930D. The virtual machine was configured as a 512MB single CPU machine with an updated version of Ubuntu 6.06, with VMware tools installed. A Python script generated 10 HTTP SSL connections (each a 10k document fetch) per second for 100 seconds. Then our script immediately paused the virtual machine, causing it to update the .VMEM file which contains the VM’s RAM. We then examined this RAM dump file for instances of words of the key in more than a dozen runs. In no cases were any words of the key found.

Effect C: Table scan fails because no identifying information. The attacker can seek to find the index table by scanning for plausible contents. Identifying the index table by its contents is difficult because: (i) the chaff is low entropy, so it can’t be easily used to find the table; (ii) the pointers in the table point to dynamically-allocated, rather than consecutive, memory addresses, so they can’t be directly used either. Examining the contents of the regions pointed to by the potential index pointers seems to be the attacker’s best approach. Some candidates can now be ruled out quickly because they point to invalid locations or locations that contain entirely zeroes. However, it remains quite difficult for the attacker to decide if a sequence of pointers actually does point to the chunk and filler, because it is difficult to differentiate a pointer to a location that contains 16 bits of scrambled key and 16 bits of filler from a pointer to any other location in memory.

Effects DD, DS: Doable with caveats. These symbols are used to mark combinations which can be accomplished but require a cost that is not expressible in computational terms. We emphasize the security of our scheme is never reliant on these factors; they are merely additional hurdles for the attacker to surpass. **DD** indicates that finding objects is theoretically possible given that they are located in RAM (and more precisely in the address space for the process that uses the key), but difficult given that they are dispersed non-deterministically by malloc(), an effect that may be enhanced by also allocating fake items of the same size. This is particularly difficult when the items have no particular identifying characteristics that readily distinguish them from other values in memory. True, in some instances, such as the chunks, they will be of higher entropy than the surrounding data, but we expect that it would be hard to pick out a single 16-bit chunk as higher entropy than its surroundings, and extremely difficult for tiny 1-bit chunks. Still, because we cannot quantify the difficulty of doing this, we must assume that it is possible. **DS** indicates that values are statically allocated by the compiler but rather difficult to find because we do not include any symbols, meaning they are simply particular bytes in the BSS segment identified only by their usage in the executable. The attacker's best attack is to rebuild the executable to find the locations.

Effect E: Run executable in emulator or virtual machine. Executable images can exploited by executing them. We believe executing disclosed memory images enables a powerful class of attacks, which have not been previously studied to the best of our knowledge. Namely, an attacker can acquire a full memory image and then execute it inside an emulator or virtual machine, where its behavior can be examined in detail, without hardware probes or other hard-to-obtain tools. Certain hardware state, primarily CPU registers, will not be contained in the memory image and must be obtained or approximated. Since operating systems save the state of the CPU when taking a process off of it, the attacker could simply restore this state and be able to execute for at least a short duration, likely at least until the first interrupt or system call. If a memory image was somehow obtained just before our prototype started loading the MMX registers with the RSA key, this basic state technique would probably suffice for the attacker to observe what values are loaded into the registers on the emulator (or virtual machine). We suspect that any obfuscation mechanism that employs software will be amenable to some form of this attack. Fortunately, we expect this attack will require significant manual work from a highly-skilled attacker.

Effect F1: Search 2^{26} possibilities for actual XOR offset and actual XOR delta. In order to interpret the index table, the attacker must circumvent the offsets and deltas, as explained in Section 2.3.3. Since these have a range of 2^{64} and 2^{32}, a brute force search requires 2^{96}. By checking each value found in memory, rather than each possible delta and offset, the search space can be reduced substantially. In this case the attacker must search each possible value from memory (M) and then compute the delta and offset that would match it on each index. That then gives a delta and offset which can be used to interpret the remainder of the table. Let $M = 1$ megabyte $= 2^{20}$. Assuming a 1024-bit key broken into 16-bit chunks, table size $t = \frac{1024}{16} = 64 = 2^6$. So that gives a total cost of $M \cdot t = 2^{26}$ for breaking the XOR offsets and deltas.

Effect F2: Search 2^{36} to determine XOR offset control value and XOR delta control value. In order to interpret the index table, the attacker must circumvent the offsets and deltas, as explained in Section 2.3.3. Assuming the attacker has somehow found the offsets and deltas in RAM, let us examine the possibility of determining the control value that specifies which offsets to use to compute the XOR offset and the control value that specifies which delta values to use to compute the XOR delta. Since the control values have a range of 2^{32} and 2^{16} (and the offsets and deltas themselves have a larger range), a brute force search would require 2^{48}. Limiting the XOR offset to a plausible set of values yields a search space of 2^{20} for the offset (i.e., only check XOR control values that result in pointer values that address within the data segment, which we'll assume is 1 M). Since the attacker needs to find the offset XOR for the pointers and the delta XOR for the chaffs, the search space is $2^{20} \cdot 2^{16} = 2^{36}$. Note that since these values cannot be verified to be correct until an RSA sign operation verifies the actual resulting key, this 2^{36} is a multiplicative factor in the computational cost of finding a key with any process that includes this step.

Effect G: Circumventing table compile-time constant ordering defense requires 2^{32}. Section 2.3.3 describes how the pointers in the index table can be permuted using a compile-time constant providing 2^{32} permutations. In order to discover the key, the attacker must try all 2^{32} permutations to see if each one gives a key that produces a correct result when used.

Effect H: Chunks encoded with 16 bits of chaff (per chunk). Each chunk is XOR'd with its own chaff (16 bits of random data). If attacker can't decode and validate a chunk at a time, brute-forcing these is clearly computationally infeasible: e.g., ${2^{16}}^{\frac{1024}{16}}$ for a 1024-bit key in 16-bit chunks. If the attacker were somehow able to validate an individual chunk, then the cost is only $2^{16} \cdot \frac{1024}{16}$, which is negligible. However, since a chunk is merely 16 bits (or even 1 bit if $b = 1$ and $s = 1$) of high-entropy data with no particular structure, we cannot conceive any way an attacker could validate an individual chunk.

Effect I: Chunks have 2^{296} possible orders. Even if the chunks were correctly decoded, they still must be assembled in the correct order to form the key. However, even for a 1024-bit key broken only into 16-bit pieces, there are 10^{89} permutations of the pieces, which is approximately 2^{296}.

2.5.3 Security Summary

The best computational attacks ("Full Disclosure" and "Partial Disclosure Untargeted" columns) require checking 2^{58} RSA keys, which costs about 10 million CPU years. If a special targeted partial disclosure attack can somehow be conceived, there is a 2^{32} attack, which takes some computation but is quite feasible. A skilled and knowledgeable attacker that has a great deal of time and patience can break the scheme with a couple of different highly-manual attacks: either reverse-engineering the particular executable on the attacked system and applying the results to the disclosed image, or setting up a carefully-timed disclosed image to be executed on an emulator or virtual machine and reading the key from the registers when they are populated.

This is a great contrast to a typical system, which is fundamentally vulnerable to Shamir and van

Someren's attacks [57] which scan for high entropy regions of memory (note keys always must be high entropy so they cannot be easily guessed) and might require checking around a few dozen candidate keys. Recall [32] showed that unaltered keys are visible in RAM in the common real systems Apache and OpenSSH. The successful attacks shown in [31] suggest that typical systems are likely also vulnerable to data-structure-signature scan methods to find Apache SSL keys and scans for internal consistency of prospective key schedules to find key schedules for common disk encryption systems.

From this analysis we see that our defenses would be especially effective against automated malware attacks, which we expect to be the most probable threat against low-value and medium-value keys. High-value keys may be worthwhile for an attacker to specifically target with manual effort, but we expect systems using those will likely use hardware solutions such as SSL accelerator cards and cryptographic coprocessors. Such hardware is too expensive for most applications, but provides high performance as well as hardware key protection for high-end applications.

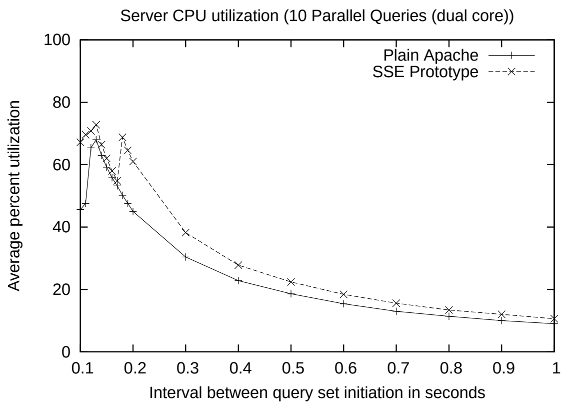
2.6 Performance Analysis of Prototype

Microbenchmark performance. First we examine the performance of RSA signature operations in isolation. Using our modified version of OpenSSL on a Core2Duo E6400 dual core desktop, a 1024-bit RSA sign operation requires 8.8 ms with our prototype versus 2.0 ms for unmodified OpenSSL. This is expected because we can't use Chinese Remainder Theorem (because we can't fit p and q into the registers in addition to d due to their space limitation). Nevertheless, our prototype just used the most basic (and therefore slowest) square-multiplication technique for modular exponentiation offered by OpenSSL, which could be improved by using Montgomery multiplication.

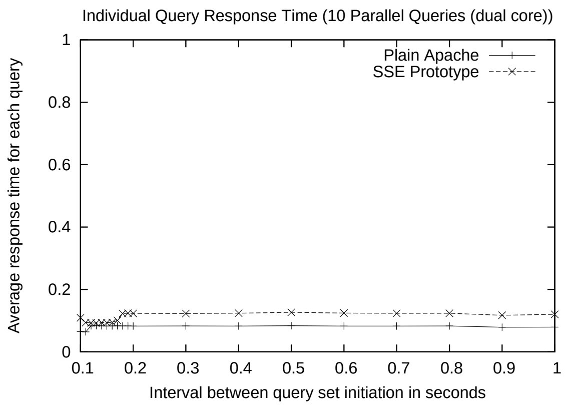
Apache Web Server SSL Performance. Now we examine the performance of our prototype within Apache 2.2.4, using a simple HTTPS benchmark. An E6400 acts as the client and another E6400 dual core desktop on the same 100 Mbps LAN acts as the server. For the first test we initiate 10 SSL connections every 0.2 seconds, fetching a ten kilobyte file and then shutting down. The 0.2 second interval was chosen because it represented a reasonable load of 50 new connections per second. We note our solution is not expected to be used for high-throughput servers, which would often use special hardware for accelerating cryptographic processing. The result is that average query latency over 100,000 requests increases from about 80 milliseconds for unmodified Apache to about 120 milliseconds for the prototype (recall all 10 queries are initiated simultaneously, which slows average response time). Average CPU utilization also increased from 45% to 61%. From this we conclude there is no substantial impact on observed performance under reasonable load, and that the throughput we measured should be sustainable over long periods of time.

In many ways this experimental setup represents a worst-case. SSL negotiation including RSA signing is done for every transfer, with no user think time to overlap with, whereas we expect real-world SSL connections transfer multiple files consecutively and have long pauses of user think time where other requests can be overlapped. Moreover, we access a single local file that will doubtless be quickly retrieved from cache, whereas we expect that real-world HTTPS interactions will frequently require a disk and/or database hit.

We also demonstrate the scalability of our prototype systems. Figures 2.5(a) and 2.5(b) show Apache server CPU utilization and response time for the 1024-bit SSL benchmark as a function of interval in seconds between sets of 10 requests, with 5000 requests per data point, demonstrating that our prototype scales



(a) Apache server CPU utilization



(b) Query response times in seconds

Figure 2.5: Apache SSL benchmark CPU utilization and response time, as function of interval in seconds between sets of 10 requests

about as well as Apache. In these experiments, the behavior of Apache becomes distorted when CPU utilization exceeds approximately 70%; the reason for this is unknown but may be because of scheduling. This can be seen in the dips and valleys on the left of Figure 2.5(a), and likely causes the similarly-timed aberrations on the left of Figure 2.5(b). Because each data point is from only 5000 requests, on a testbed which is not isolated from the department network, there is some noise which causes minor fluctuations in the curve, visible on the right of Figure 2.5(b).

2.7 Conclusion and Open Problems

In this chapter we presented a method, as well as a prototype realization of it, for safekeeping cryptographic keys from memory disclosure attacks. The basic idea is to eliminate the appearance of a cryptographic key in its entirety in RAM, while allowing efficient cryptographic computations by ensuring that a key only appears in its entirety in certain registers.

Our investigation inspires some interesting open problems such as the following.

First, our work focused on showing that we can practically and effectively exploit some architectural features to safekeep cryptographic keys from memory disclosure attacks. However, its security is based on heuristic argument. Therefore, it is interesting to devise a formal model for rigorously reasoning about the security of our method and similar approaches. This turns out to be non-trivial partly due to the following: If an adversary can figure out the code that is responsible for loading and resembling cryptographic keys into the registers, the adversary would still possibly be able to compromise the cryptographic keys. Therefore, to what extent we can say at which degree the adversary can reverse-engineer or understand the code in RAM? Intuitively, this would not be easy, and is related to the long-time open problem of code obfuscation, which was proven to be impossible in a very restricted model in general [6]. However, it is open whether we can achieve obfuscation in a less restricted (i.e., more practical) model.

Second, due to the limitation of the volume of the relevant registers, our RSA realization was not based on the Chinese Remainder Theorem for speeding up modular exponentiations, but rather the traditional "square-multiplication" method. This is because the private key exponent d itself occupies most or all of the XMM registers. Is it possible to circumvent this limitation by, for example, designing algorithms in some fashion similar to [9]?

Chapter 3

Securing Digital Signing with the Protected Monitor

3.1 Introduction

It is now well-accepted that cryptographic schemes deployed in real-life systems should be accompanied with rigorous security proofs. Digital signatures are a widely used cryptographic tool for assuring various authenticity: non-repudiation and sources of data access control requests (while possibly protecting privacy if desired [16]); sources of data items or software programs in the form of provenance for evaluating their trustworthiness [?, 76, 69].

However, there is a gap between the authenticity offered by digital signatures in the abstracted models (see [29] for the classic and standard definition) and the authenticity required by real-world applications. This is because the abstracted models (inevitably) have to assume away some attacks that are relevant in a broader security context otherwise. A particular type of such attacks was called *hit-and-stick* but left as an open problem in the literature [71]. In this attack, the attacker (via malicious stealthy malware) penetrates into a computer system, while possibly evading current security mechanisms. As a consequence, the attacker can compromise the private signing keys, and/or compromise the private signing functions by simply feeding whatever messages to the program/device that holds the private signing key. A prototype of such attacks may be found in [33], while the concept was already highlighted many years ago in a seminal paper by Loscocco et al. [43], who also highlighted that special hardware devices are no panacea (because the attacker can compromise a private signing function *without* compromising a private signing key).

Our contributions. In this dissertation we address the hit-and-stick attack against digital signatures in real-life systems. Specifically, we present the design of a general, extensible framework for enhancing the authenticity offered by digital signatures (Section 3.2.1). The framework offers digital signatures with systems-based assurance that can be verified by the signature verifiers, which is very useful in application such as analyzing the trustworthiness of data via their digitally signed provenance. The framework utilizes both trusted computing and virtualization simultaneously. It is extensible because it can integrate other virtualization-based security mechanisms so as to fulfill a more comprehensive security solution (rather than just for protecting cryptosystems).

Further, we present a concrete design, implementation and evaluation of a light-weight system as a prototype instantiation of the general framework (Section 3.2). The core of our solution is a novel software module called *protected monitor*, which is a light-weight software substrate beneath guest OS kernel but residing on top of the hypervisor, and might be of independent value. Our in-VM protected monitor is much more powerful than VM introspection because it largely solves the semantic gap problem, created by attempting to understand the semantics of operations of a virtual machine from outside the VM. Moreover, our solution has several features. First, private signing keys are not directly accessible from the user's (compromised) VM, even via raw disk access, meaning that a malware can no longer easily disclose the keys. Second, our solution does not require modification to the application source code, which greatly increases its applicability. Third, applications requesting use of the key can be attested before they are allowed to use the key.

In addition to providing experimental performance evaluation (Section 3.4), we conduct a systematic security analysis against a number of possible threats against the system, which shows that the resulting system has no security flaws as long as the underlying hypervisor is secure (Section 3.3).

3.2 Assured Digital Signing

Objective and design requirements. The objective of assured digital signing is to add systems-based security assurance to the cryptographic properties of digital signatures so that we can get the best of both worlds — systems security and cryptography. As a result, the signature verifier can have better trustworthiness in the data the signature vouches for, which is important in the verifier's decision-making process. Our proposed framework is to accompany a digital signature with an assertion on the system state under which the signature was generated. The framework is general in the sense that it can accommodate many other specific techniques for monitoring the state of the system and can be integrated into a large class of security mechanisms for a comprehensive solution. Moreover, it can accommodate the architecture that already offers a hardware device for conducting cryptographic computation. Communication layers are provided to make inter-VM communication transparent to the application, which is still written as if it is invoking the CSP as a local library. In what follows we explore the design space while bearing in mind the above requirements.

Why hardware/TXT alone is not sufficient. In order to defeat the threat of software-based attacks against private signing keys, we can certainly store them in some hardware devices such as co-processor [74] or Trusted Platform Module (TPM) [30]. However, it is much more difficult to defeat software-based attacks that target the private signing functions rather than the private signing keys. This is because once the attacker penetrates into and compromise the Operating System (OS), to which the hardware devices are attached, the attacker can simply asks the hardware devices to sign any message it likes. The same attack disqualifies both Intel's Trusted Execution Technology (TXT) technique [35] and AMD's Secure Virtual Machine (SVM) [2], which provides a hardware-protected clean execution environment on-demand (i.e., without re-booting the system). This is because the invocation of such an environment is realized through a privileged instruction, which however can be launched by the malware that has compromised the OS kernel already. As a consequence, this also disqualifies the follow-on solutions that exploits the TXT technique (e.g., [45]).

One obvious countermeasure to this attack would be to deploy CAPTCHA system [13], namely by challenging the digital signing requester to solve some problems that can only be solved by human. However, this solution is either annoying because the requester has to solve a CAPTCHA challenge for every digital signing, or not possible because the signing process is invoked automatically by other applications programs (i.e., without involving human in the loop). More importantly, this solution is actually not secure because the attacker, who has compromised the OS and controlled the communication channel between the user and the hardware device, can launch the following man-in-the-middle attack: The attacker can simply utilize the user to help it solve the CAPTCHA challenge and then prompt to the user that for the last time the entered solution to the CAPTCHA challenge was incorrect. The user would not suspect there is a man-in-the-middle attack because as human we might often make mistakes in discerning or typing solutions to CAPTCHA challenges, which is especially true as the CAPTCHA is getting more and more sophisticated so as to defeat automatic CAPTCHA solvers [13]. The use of TXT-based trusted I/O may be able to defeat this man-in-the-middle attack because the malware cannot incept the user's input. However, this approach has the drawback that everything else running on the system has to be frozen in order to run the system. Moreover, as mentioned above, no human is involved in many applications and thus disqualifies this solution.

3.2.1 System Logical Design

In the above we have discussed why hardware/TXT alone is not sufficient to defeat the hit-and-stick attack against digital signing. The cause of this phenomenon is that the attacker can penetrate into the OS of the victim computer and thus impersonate the user or user program. The ideal solution to this problem is to ensure that the OS is never penetrated, which is however a grand challenge that might remain open for decades. As a practical and feasible solution, we would have to make some assumption that there are some small Trusted Computing Base (TCB) in the software stack. This leads to the architectural framework depicted in Figure 3.1, where the small TCB is naturally realized by the hypervisor. We assume the hypervisor is secure, which is an active research topic [62, 67, 5].

Capturing dynamic system properties is an important yet challenging research problem that remains to be tackled [38, 72]. Our approach is orthogonal to efforts in this one because we can take advantage of them in a plug-and-play fashion. This also applies to research that aims to ensure kernel integrity and detect kernel rootkits [44, 15].

Attack hypervisor. Lastly, the attacker could seek to attack the hypervisor. Since our attack model is that the attacker is working from code running within the user VM, we expect this to be quite difficult. We do not know of any successful attacks against production-quality hypervisors such as Xen and VMware from within a VM. If there were some way to attack the hypervisor successfully from within the User VM, then the Secure VM could be compromised and its disk could be read.

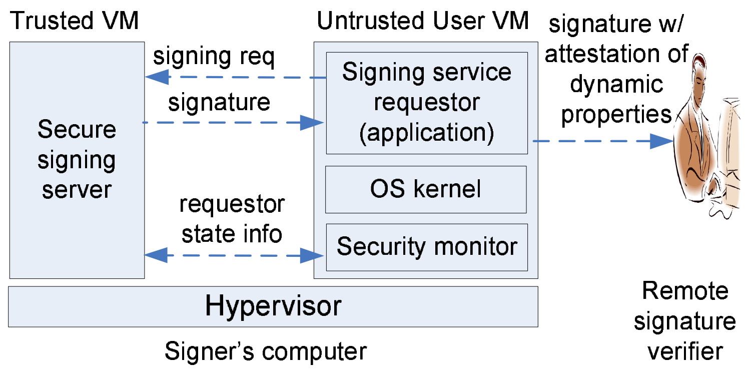


Figure 3.1: Logical design of solution framework to the hit-and-stick problem (dashed arrows represents logical, rather than physical, communication flows)

In this framework, a signature verifier verifies not only the cryptographic validity of a digital signature (i.e., the signature is valid with respect to the claimed public key that was not revoked), but also the attestation about the system environment in which the signature was generated. Because we want to prevent, rather than just detect after the fact, attacks against the signer's computer, the attestation ideally should include state information such as whether the system (especially the application program that issued the signature) is under attack or suspicious. Correspondingly, the signer's system, which needs to collect the relevant information, is characterized as follows. We separate the applications from the signing server because we want to make our solution extensible so as to integrate with other existing and to-be-developed solutions. We note that it is relatively easy to protect the cryptographic server than to protect the cryptographic service requester because the former is almost always static, whereas the latter resides in a system that often needs to be updated with new software programs or their patches. This justifies why we use a trusted VM for the actual signing program, while the application runs in an untrusted VM. This allows to integrate with existing and future VM-based introspection solutions (such as those mentioned above) for a more comprehensive solution. Moreover, the use secure monitor with the user's untrusted VM, which could be integrated with other in-VM introspection security mechanisms. This is appealing because in-VM introspection has certain advantages over out-of-VM introspection. The security monitor is not designed to be a part of the TCB because we want to make as few changes to the TCB as possible. The security monitor is a security-critical module that should reside directly on the TCB. Moreover, the security monitor can integrate existing countermeasures against the compromise of the requester software.

3.2.2 System Design

Figure 3.2 depicts the the overall physical design of our signer system. We choose Xen as our platform because the code is freely available. The physical design details many issues that were abstracted away at the logical design mentioned above. In what follows we elaborate on the main components in the system.

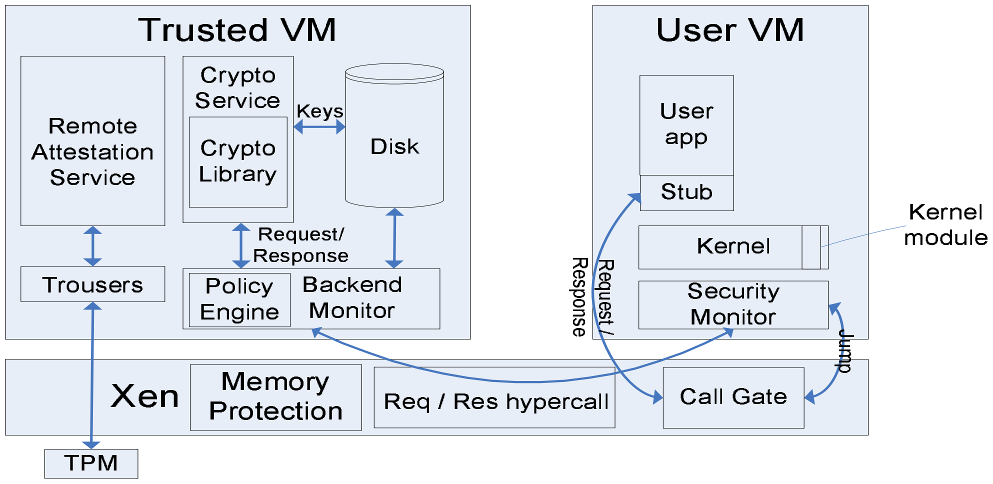


Figure 3.2: Overall software architecture of the system

3.2.2.1 System Components in Xen

The relevant mechanisms that our system needs support from the hypervisor are: memory protection, request/response hypercall, and call gates. In the below we elaborate them.

Memory protection. Xen-enabled memory protection is a key component of our security, because it allows us to protect data and code from modification by an attacker in the domain U. Most importantly, we need to provide memory protection for the Security Monitor in order to protect the security monitor from being compromised.

A trusted VM runs a Remote Monitor that can persist information. (Note that hypervisors typically have no ability to persist data, and the file store in the User VM cannot necessarily be trusted.) Highly secure code that authors do not wish to port to run within the monitors themselves can also be run within the trusted VM, allowing them to make use of ordinary operating system services.

We modify the Xen hypervisor to add memory protection for the Protected Monitor that sits within the User VM, and also to augment Xen to allow inter-VM communication without having to rely on the user VM kernel and operating system in any way. The Protected Monitor within the user VM can handle simple access control decisions without having to cross the VM boundary. More complex decisions, including decisions that are best made outside the VM, are sent to the Monitor in the Secure VM, which will be in Xen's Domain 0.

User applications are run inside the User VM, where the protected monitor has been inserted above the kernel. Our protected monitor can be seen as superior to the kernel, meaning that the protected monitor is not only difficult to attack, but could be used to mediate kernel actions if desired. The protection is achieved by using virtual machine page protections to protect a region of kernel memory where our protected monitor will reside. This memory is protected against execution and modification, except during a special mode that only applies when the monitor is executing.

Request/response hypercall. We extend Xen's hypercall mechanism to provide several additional hypercalls to support our system design. These are ... **Weiqi, what are the new hypercalls?**

Call gate. We utilize the call gate mechanism provided by x86 hardware in order to escalate privilege from ring 3 (user mode applications in dom U) to ring 1 (the ring level for the domU kernel and our security monitor). The unique feature of the call gate mechanism is that we can raise the privilege level without modifying or using the kernel of its data structures. Normally the kernel would control access to the Global Descriptor Table (which specifies call gates and other system descriptors) but in a Xen system this access is controlled by Xen. So we changed the Xen code and tables that initialize this table. **Weiqi, did we have to make any other changes to Xen related to the call gates?**

In theory privilege escalation could be achieved using system calls or hypercalls, which we use in other places and are more typical mechanisms for escalation. Using call gates rather than system calls or hypercalls has the following advantages:

- Allows us to hash the dom U application and know we have the correct process, since read process CR3 directly from it.
- Prevents CR3 or page table modification by attacker; we use the CR3 and page table the process is actually using.
- Allows us to know we actually invoked Xen (because Xen controls access to the GDT), rather than some program in domain U pretending to be the hypervisor.
- Communication via call gate 2 ensures kernel can't selectively block messages. (I.e., if sent messages via kernel module, kernel could selectively block some messages.)

3.2.2.2 System Components in Domain U

The main system component in the user domain is a new substrate we call security monitor. The function of the protected monitor, which is a core part of the system, is to allow userspace domain U applications to communicate directly and securely with domain 0. The main issue encountered in the design of the protected monitor is to enable communication between domain U and domain 0 *without* the support of kernel.

Stub. The stub layer automatically marshalls and demarshals cryptographic library calls and forwards the calls to domain 0, providing transparent access to the service provider in domain 0.

The stub exists to allow the user application to transparently invoke what appear to be ordinary library calls. However, instead of the request being processed inside the local library, it automatically translates them into requests that travel via the security monitor to be served by the service provider in domain 0. The stub code declares functions in the crypto library so that the user code can link against them just like linking against a static or dynamically-linked implementation of the crypto library. Since the definitions of the functions accept the library arguments and marshal them appropriately and send them to domain 0 which then processes them and then the stubs deserialize the reply, the user application is completely unaware that the operations are not implemented directly in the library.

Kernel module. The kernel module enables user processes to invoke certain hypercalls, since user processes cannot invoke hypercalls directly. Invoking the kernel module is also faster than using invoking a call gates, so best to not use gates for everything. The downside of using a kernel module is that a compromised kernel could prevent it from operating. For this reason we never use the kernel module for security-sensitive operations, only to set up and tear down the system. If those operations fail, the result is merely a denial-of-service.

Security monitor. When an application process in Domain U requires a cryptographic service, it invokes the cryptographic service provider stub. The stub uses a call gate to invoke an appropriately marshalled hypercall (including the identity of the specific function that is requested) so as to send a Xen event across across a channel to the secure VM. Note that some privilege escalation must be done by Xen hypercalls rather than the call gates or invoking the kernel module. This is because hypercalls are the only way to communicate with hypervisor. Note that due to the design of Xen (and other typical hypervisors), hypercalls can only be invoked directly from code in the kernel or a kernel module, so we could not implement communication from userspace securely and efficiently using only hypercalls.

3.2.2.3 System Components in Domain 0

The system components in the trusted VM include: (i) backend monitor, (ii) remote attestation service, (ii) crypto service, (iv) disk. Below we describe the components in detail.

Backend monitor. This is the counterpart to the protected monitor inside the trusted VM. It has 3 major functions: facilitating communication (see Section 3.2.2.4), determining which communication requests to approve or deny (the policy engine), and inspecting the domain U caller generating a request. The most complex job of the backend monitor is inspecting the domain U caller. The backend monitor has three primary responsibilities:

- Facilitating communication. Upon receiving the event, Xen maps in memory pages that were transferred to the secure VM in order to read the marshalled function number and arguments. A stub layer for the cryptographic service provider will recreate the actual C language invocation from that data. Backend Monitor in Domain 0 receives virq and translates them into appropriate user-level library invocations, which requires unmarshalling the arguments.

- Policy engine. The goal of the policy engine is to allow the creation of flexible policies for approving and denying requests made via the remote monitor, based on decision criteria available to the remote monitor, such as whether hash values match. This is relatively straightforward from a coding perspective and we did not implement it.

- Inspecting the domain U caller. In order to establish the authenticity and integrity of an executing program that claims the right to use a certain key, we need to authenticate the caller. A typical solution for such a problem would be to compute a hash of the executable image in memory. This presents two problems: how do we know what the hash of an executable should be, and how do we deal with the need for updates to an executable, which will change its hash? For a program that does not ever change the answer is simple enough: if the hash of the program that requested generation of the key is the same as the hash of the program that requested use of the key, then the use should be permitted. For the more common case of a program whose executable is periodically updated, a more sophisticated mechanism is required. Here we introduce the concept of the provenance of an executable. By this we mean establishing a trail that establishes how an executable was obtained or from what source it originated. When an application initially creates a key, we compute a hash of the executable and check it against a signature provided by the publisher. If the signature matches, we then record the publisher as having the right to produce future applications of the same name that can use this key. Neither applications from other publishers nor other applications from this publisher have the right to use the key.

It's important to note that we hash the executable at the first call gate, and then lock the executable pages so they can't be modified. This is for two reasons: (i) The performance impact is lower, since there is a hash at the beginning instead of every time a message is sent. (ii) This prevents subtle TOCTOU attacks which would otherwise be possible (e.g., changing the binary just before sending the message, then somehow changing it back afterwards).

Some technical issues need to be resolved in order to compute this hash. First, we need to know what comprises the executable, while avoiding any dependency on the kernel as far as possible. Second, we wish to perform this operation efficiently since a process could have a large set of pages. In the end we chose to examine the pages in the user process code segment. This gives us the executable and all libraries, including shared libraries, while avoiding any reliance on the kernel or its data structures and still giving better performance than other options.

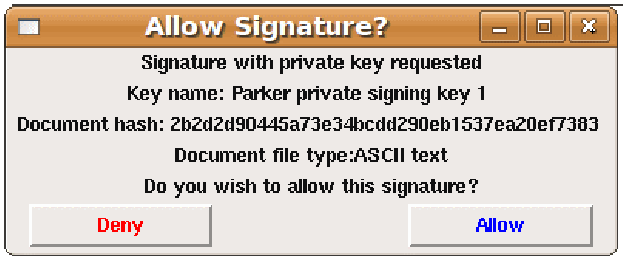
Crypto service. This component provides the cryptography service. It consists of two pieces: the crypto library itself and a wrapper which enables the library to receive calls made across the VM boundary.

Disk. This is simply the ordinary disk in domain 0. Note that there is no way for the domain U to access the domain 0 disk, so any information on the domain 0 disk is secure from the domain U.

The disk is important because it stores the keys used by the crypto service, as well as the implementation of that service and all other domain 0 software components.

Attestation service. Attestation in our concrete implementation includes the following: (i) static measurement of boot and kernel (using TPM); (ii) secured crypto library; (iii) authentication of the requesting program (measure binary); (iv) trusted path user confirmation dialog.

Figure 3.3 depicts the optional trusted path user confirmation dialog. This runs from domain 0 so that it displays directly on the console using X Windows and enables the user to explicitly approve each signature request made with their key. While our prototype simply records the bytes of the message and shows the corresponding file type, a full implementation could feed the bytes to a document viewer so that the user could see the actual document being signed (if it is of a type that is a viewable document). Because this dialog is part of the domain 0 service provider code, its operation is completely transparent to domain U, which is completely unaware of its existence, except for a couple of changes in the behavior of the signature request call: 1. the call does not return until the user indicates their decision, and 2. well-formed signature requests will fail if the user disapproves the request.


Allow Signature?

Signature with private key requested

Key name: Parker private signing key 1

Document hash: 2b2d2d90445a73e34bcdd290eb1537ea20ef7383

Document file type:ASCII text

Do you wish to allow this signature?

Figure 3.3: User signature confirmation dialog (optional)

Trousers. Trousers is an open-source implementation of the TCG Software Stack (TSS), created and released by IBM. This enables domain U applications to access the TPM using the software API designed by the Trusted Computing Group.

3.2.2.4 Putting the Pieces Together

Shared memory and communication flow. Shared memory is the mechanism we use for efficient communication between the Trusted VM and the User VM. By mapping the same pages into both VM's, messages can be sent from one domain to the other without any copy operation, making message transmission a fixed cost irrespective of message size, which is important since users may request signatures on large amounts of data. In order to guarantee the security of the security monitor. We allocate 1024 physical pages of memory (4MB). The first 256 physical pages (M_0) contain the wrapper function that used to invoke hypercall to request Trusted VM giving the crypto service. The second 256 physical pages (M_1) are used to store the measurement of user VM application's code segment, user VM system call table, IDT, and parameters. The third 256 physical pages (M_2) used for user VM application write the message that need to sign. The last 256 physical pages (M_3) used for Trusted VM to write the result.

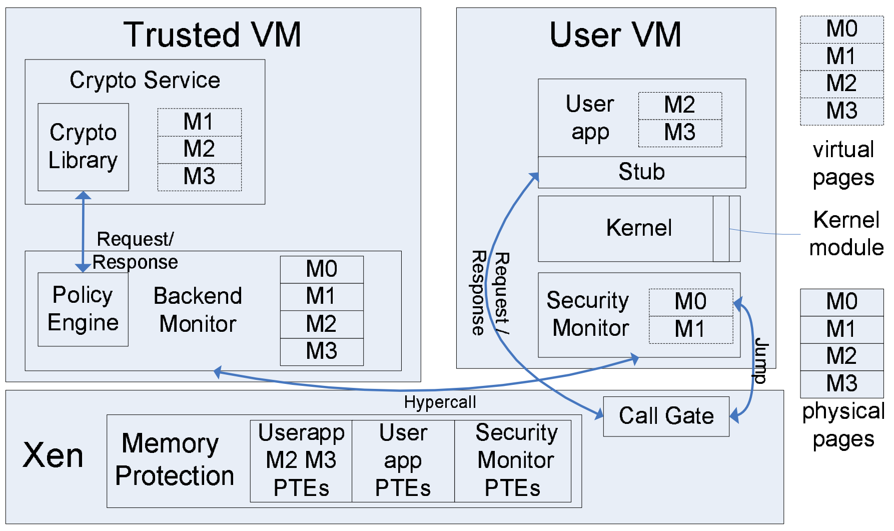


Figure 3.4: A high-level description of the system shared memory

There three parts of virtual pages map to the physical pages: (i) In user VM's kernel space the security monitor maps the M_0 and M_1. (ii) In user VM's user space the user application maps the M_2 and M_3. (iii) In Trusted VM's user space the Crypto Service maps the M_1, M_2 and M_3. Because (i) & (ii) are both in the User VM, the page tables of these virtual pages need to be protect by memory protection in Xen.

Recall that our design goal is to require no code changes for the user applications, so we simply relink it against a stub library, which is particularly easy if the application is dynamically-linked. This stub library must achieve a communication layer where inter-VM communication is completely hidden from the ordinary user-space pplication, which is still written as if it is invoking the CSP as a local library. In order to fulfill secure kernel-free communication without making any modifications to the Domain U OS kernel, we need to realize privilege escalation as follows.

Figure 3.5 summarizes the steps in system execution, with emphasis on message flow between entities. During the preparatory step, kernel modules are loaded in domain 0 and domain U. It is important to note that the domain U kernel module is not used to implement any security-sensitive functionality. If the domain U kernel blocked it or blocked some of its functionality, it would be able to achieve only a denial-of-service attack. All interrupts are sent and received through `ioctl` operations on the device files that are the interface to the kernel modules. Here are the actual system steps as executed under the direction of user land applications in domain 0 and domain U:

1. The kernel module devices are opened, which causes them to register themselves to handle certain virtual interrupts (software interrupts generated by Xen).

2. Domain U uses the kernel module to request that the hypervisor send an interrupt to domain 0. This

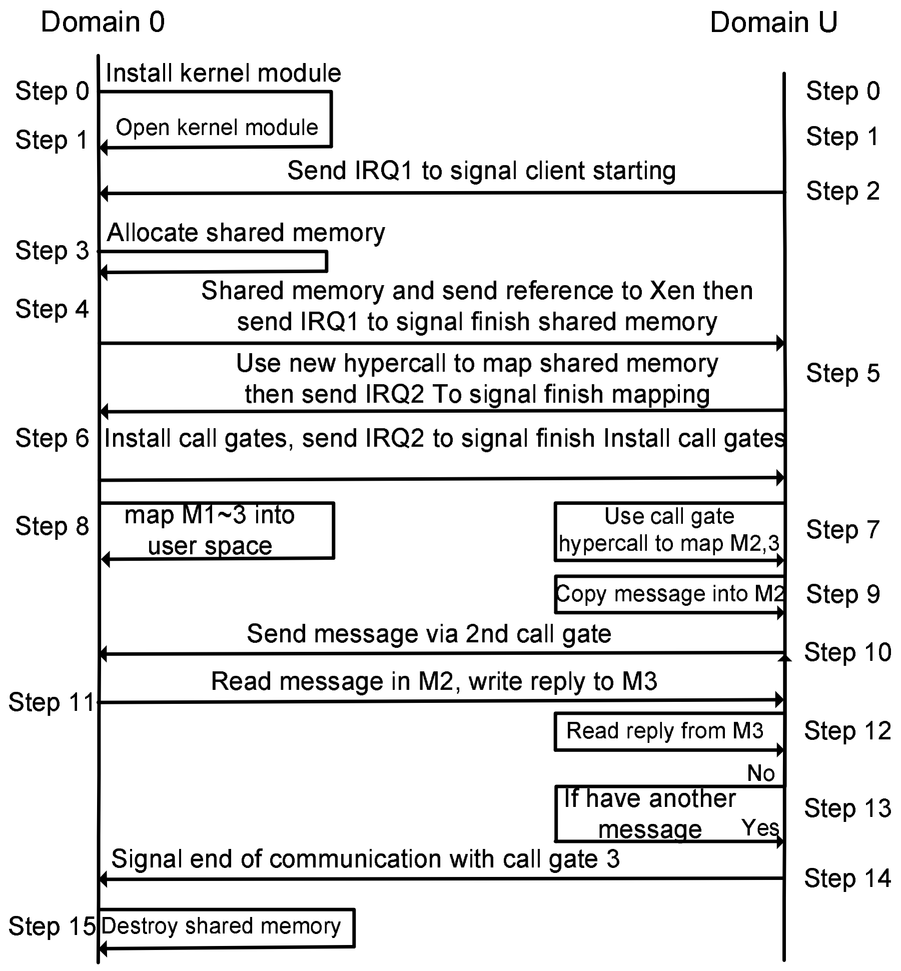

Domain 0
Domain U
Step 0
Install kernel module
Step 1
Open kernel module
Step 0
Step 1
Send IRQ1 to signal client starting
Step 2
Allocate shared memory
Step 3
Step 4
Shared memory and send reference to Xen then send IRQ1 to signal finish shared memory
Use new hypercall to map shared memory then send IRQ2 To signal finish mapping
Step 5
Step 6
Install call gates, send IRQ2 to signal finish Install call gates
Step 8
map M1~3 into user space
Use call gate hypercall to map M2,3
Step 7
Copy message into M2
Step 9
Send message via 2nd call gate
Step 10
Read message in M2, write reply to M3
Step 11
Read reply from M3
Step 12
No
If have another message
Yes
Step 13
Signal end of communication with call gate 3
Step 14
Step 15
Destroy shared memory

Figure 3.5: A high-level description of the system control flow

interrupt, ”irq1”, is used to signify that a client is starting up.

3. When the domain 0 application receives this interrupt, it allocates 4 megabytes of memory.

4. The domain 0 application then shared memory and uses two pages to store the 4 megabytes of shared memory’s reference(share memory each page has a reference 1024 pages so 1024 int reference, later step 5 our new hypercall can use these to map 4 megabytes) and mfn(for later step 5 to protect domU map the shared memory). And then uses a hypercall to send the location of the shared memory and the pages of reference and mfn to Xen. So that domain U can use the our new hypercall to map this memory.

5. After waiting on IRQ1, Domain U invokes a special hypercall to map the megabytes into kernel user address space for the domain U application. (Mapping the memory into its address space is what allows domain U to “share” this memory with domain 0.) This hypercall is different from the one

ordinarily used to share memory in Xen, because of our special security requirements. This memory is read-only and "map-protected," which prevents it from being mapped using normal Xen sharing hypercalls. This is discussed in more detail in Section 3.3. Domain U then indicates that it has finished mapping the shared memory by sending IRQ 2.

6. When domain 0 receives IRQ2, it modifies the GDT (the x86 Global Descriptor Table) to install the call gates for use in Domain U. It then installs the wrapper function in M_0, where it will be invoked by domain 0, and exit_page in the last page of M_1. The last page of M_1 is always write-protected, so that domain U cannot write it. M_0 cannot be written from domain U normally, but becomes writable while Domain U is inside call gate 2. Domain 0 then sends IRQ2 to domain U to indicate that the call gates are set up.

7. When domain U receives IRQ2, it can then invoke the first call gate. The effect of the call gate is to raise the CPU privilege level to ring 1 from the user-level of ring 3 and begin executing code at a specified location. The application binary is also measured (hashed), and the executable pages of the application are locked (and marked read-only if not already) so that they can't be changed. At the same time, the CR3 and page table in use do not change, so hypercalls can be made directly from the user application and operate on the page table of the user application.

   In our case, the call gate is set to execute code in M_0. For call gate 1, this code simply invokes a hypercall (which is otherwise not possible without going through the kernel). This hypercall is used to map M_2 and M_3 memory into the user process. M_3 is set read-only and M_2 can be read or written.

   As soon as this returns, the user application can put a message into M_2 whenever it desires.

   Secondly, a hash is computed for the domain U application that invoked the call gate, from the Xen hypervisor. Because the call gate process retained the CR3 and page table of the process, this uses the page tables of the process, which prevents various attacks that try to substitute different code when a process is being hashed. By using the page table of the process, we can ensure these are the same pages the process would actually access and execute. The executable pages of the application are then marked read-only (if they weren't already) and Xen is informed to protected the PTE's of the executable pages, so that the kernel can't modify the page table to point at different pages. I.e., the pages themselves cannot be changed, and the VM subsystem "pointers" to the pages cannot be changed.

8. In the meantime domain 0 maps M_{1-3} into user space. So that domain 0 user space can easy get the hash of domain U system call table, IDT and userapp executable pages in domain U. And also the

two parameters from domain U.

9. The user application copies its message into M_2.

10. Domain U then invokes the second call gate, which means to send the message in M_2. Invoking this call gate runs the code in shared memory, which has two major steps: First, a single Xen hypercall is made, and then (i) M_2 is marked as not writable. This is a second way to ensure that domain U cannot interfere with the message being sent; we had already ensured that the kernel could not write it and that it was only accessible by the process [1]. [2] [3] (ii) $M_{0,1}$ is marked as writable (except the exit_page, the last page in M_1, which remains executable but not writable). (iii) The process hash, a, and b are recorded in M_1.(hash of domain U system call table and IDT also recorded in M_1) (iv) IRQ2 is sent, informing domain 0 there is a message waiting to be processed. (v) Domain U then waits for irq3, signifying the message has been processed and a reply is available. Second, we execute exit_page. This transitions us back to user mode after invoking a hypercall that makes $M_{0,1}$ read only.

11. When domain 0 receives IRQ3, it knows there is a message available, so it reads it from M_2. When a reply is ready, it places the reply in M_3 and sends IRQ3 to let domain U know the message has been processed and a reply is available.

12. When domain U receives IRQ3, it executes a hypercall to make M_2 writable again in case it wants to send another message. It reads the reply from M_3.

13. If domain U wishes to send another message, it returns to step 10. Note that domain U needs to send a termination message, because domain 0 has no way to know otherwise when the connection should be torn down.

14. When domain U has finished with all messages it wants to send, it invokes call gate 3. This unmaps $M_{2,3}$ and sends IRQ3 to domain 0, to inform it that it is no longer attached to the shared memory. Domain U then unmaps M_{0-3} and closes the device file that connects it to the kernel.

15. When domain 0 receives the IRQ3, it unmaps $M_{2,3}$, closes the device file that is connected to the kernel module, and destroys the shared memory.

[1]Weiqi, what about other processes? I assume that right now it just breaks if we try to set up communication for a second process while one is already communicating? What do you think we should say here?

[2]Weiqi: please double-check this; I replaced this footnote entirely already thinking what what you wrote in your reply. ‘Note this is a second defense because there is potential for a race conditions. The race condition occurs if the process received a software interrupt that caused some malicious code inside the process to execute after writing the message but before invoking the call gate. In this case we would have detected the corrupted executable when we measured the executable. For extra protection the messaging code could temporarily disable software interrupts, (e.g., with `sigprocmask()`).’

[3]Weiqi ask: Because domain U kernel has M_0 M_3, userapp has M_2 and M_3. Domain 0 kernel has M_0 M_3, sender.c has M_1 M_3. How to let reader know M_0 is whose? Paul reply: I’m not sure I understand your question. Are you talking about preventing race conditions for writing M_0, or something else? If so, when does a race condition occur?

3.2.3 Implementation

Our system was implemented in the following environment. The hypervisor is Xen v3.3.1. Domain 0 runs Ubuntu 8.04 (Linux 2.6.18.8-xen.hg kernel) as its guest OS, and domain U runs Ubuntu 8.04 (Linux 2.6.18.8-xen.hg kernel) as its guest OS. For the digital signing library, we use Peter Gutmann's `cryptlib` library, which is available under both open-source license (Sleepycat, which is GPL-compatible) and a commercial license for closed-source commercial use. The `cryptlib` also provides certificate management services, including key generation in response to certificate requests.

3.2.3.1 Implementation of Runtime Memory Protection

In order to safely and efficiently implement the runtime memory protection of the security monitor, we did the following:

First, we ensure that the shared memory cannot be unmapped, remapped, or mapped partially via hypercalls from domain U. In order to achieve this goal, after the domain 0 shared the 4 megabytes, we fill the protect shared table with the references of these 1024 pages. Then we set the flag `shared_memory` = 1 (domain 0 already shared memory, so the domain U cannot use the normal hypercall to map these memory pages). When the domain U want to map the shared memory we check in function `gnttab_map_grant_ref`: we will see if `shared_memory` == 1 or `shared_memory` == 2 and the reference that domain U want to map is in our protect shared table or not. If is in the table we will prevent it to map this page. So the shared memory cannot be mapped partially via hypercalls from domain U.

The only way to map these memory pages is using our new hypercall. Our new hypercall just accept only two inputs: one is map or unmap flag, if the input equals to `GNTTABOP_map_grant_ref` that means map the 4MB one time. The other input is the domain U `shared_pages_addr`, this one will store in Xen. Late the we will modify the GDT to let the address point to this one. In this hypercall if the shared_memory == 1, then we begin to map. Before each page we map we'll temporarily set the shared_memory to 0 to let the normal map progress, then set the shared_memory flag to 2. To prevent domain U kernel using this hypercall to map the memory to another virtual address. And this time the domain U may using the normal hypercall to unmap the memory so we also check in the function `gnttab_unmap_grant_ref`: we will see if the `shared_memory` == 2 and the reference that domain U want to unmap is in our protect shared table or not. If is in the table we will prevent it to unmap this page. So the shared memory cannot be unmapped, or unmapped partially via hypercalls from domain U.

There are two more things we need to take care of. One is the write access: we don't want the domain U kernel to write other things like the attack code in the shared memory, so we need to make sure the shared memory is readonly. The other is the NX bit. Our original implementation interfered with the use of the NX bit during system development (32-bit PAE kernels use the NX bit), and were able to work around it, so that we didn't remove this important defense. In order to achieve these two properties, we use `*((unsigned long *)pl1e)` &= `0xfffffffd` to change the page table entry to make the shared memory in domain U are readonly; and use `*(((unsigned long *)pl1e)+1)` &= `0x7fffffff` to change the M_0 (protected monitor) and last page in the M_1 (exit page) to let these page can be executable. So that later the callgate can jump to the M_0 (protected monitor).

After the user application finishes sending all of its messages, it will ask the domain U kernel module to unmap the shared memory using our new hypercall with input `GNTTABOP_unmap_grant_ref`. So if the `shared_memory` == 2 means we can use our hypercall to unmap the 4MB shared memory one time. [4]

Second, we modify the GDT to let the callgate point to the correct address. First we need copy the protected monitor code into M_0 in domain 0. Because the domain 0 is secure VM, so the protected monitor is correct and as the write access of M_0 in domain U is readonly, so we don't need worry about the protected monitor. And the exit page code into the last page of M_1. After this we use a hypercall to modify the GDT because only Xen can change this. We add a new GDT entry during the Xen boot initialize GDT which type is callgate. But at that time we don't know the shared memory address in domain U. So here we fill the GDT entry with the domain U shared memory address.

Third, we check that the Domain U grant parameters are correct, to prevent an inaccurate or partial map request to Xen.

Fourth, Domain U grant address **Weiqi, what is this one?? Just checking the grant address? If so, shouldn't it be part of the one above?**

Fifth, we write-protect access to the shared memory during call gate 2.

Sixth, we ensure the kernel must be prevented from modifying page table entries that point to the shared memory. This is difficult because domain U kernel has three ways to modify the page table entry: hypercall `do_mmu_update`, `do_update_va_mapping` and `ptwr_do_page_fault`. And each way have two kinds of attack: one is change its own PTE to map the protect page (shared memory), the other is change the protect PTE to map other memory or change the write access bit. We discussed how we protect against these in Section 3.3.

Seventh, we ensure that kernel cannot modify M_2. For example, the kernel cannot change message after written to M_2 but before call gate 2 invoked to send the message.

Eighth, while protecting the Domain U executable and the 4M shared memory is necessary, searching

[4]Weiqi: Are you saying set shared_memory = 2 to unmap? Or are you saying unmapping requires that shared_memory == 2 already?

?????? [5] every time domain U asks to modify the page table entry to see if this is a page we need to protect would be very slow. So we used one bit in `page->u.inuse.type_info` (in Xen's `frame_table`), which we named `PGT_entry_protected`, to mark whether this page needs to be protected. So every time we merely need to check this bit, and if it is set then we prevent domain U from changing the page table entry.

3.3 Security Analysis

Here we analyze the security of a system designed as described and carefully implemented.

We consider hit-and-run and hit-and-stick attacks that can compromise the user VM. There are two basic ways to attack the system: (i) attacking the security monitor; (ii) attacking the crypto service via attacks against cryptography, or attacks against key secrecy, or attacks against applications that request digital signatures, or attacks that falsely request digital signatures. In what follows we argue why the attacks cannot succeed. We organize the analysis by attacks against components organized by their physical location: Domain U, Domain 0, and components that are not contained within a specific domain, after introducing our threat model.

Threat Model. We use the same standard assumptions typical in virtualization security architectures ([52, 26, 25, 37]):

- the hypervisor and trusted VM (domain 0) are in the trusted computing base (TCB).
- the guest VM is not in the TCB. Therefore, malicious code can only affect the TCB. (Remember that we are concerned with malware, so if all user actions are contained within the guest VM, all the malware resulting from those actions will be also. For malware that actively attacks systems externally, such as worms, we have to rely on security of the trusted VM, which is its own area of study (**Dr. Xu, what should we cite here? I added this whole assumptions section**) and is facilitated significantly by not running user code in the trusted VM.)
- the hypervisor is ideally a small layer that is both secure and verifiable, and provides isolation between the trusted VM and the untrusted VM.

Additionally, we assume that user does not install and run applications in the trusted VM, but performs all user activity in the untrusted VM. One way to achieve this for most users is to simply make the untrusted VM not be easily accessible.

This threat model is realistic, as it assumes the attacker can do anything he desires to the guest VM, including inserting both user-space and kernel-space malicious code.

[5] Weiqi: What data structure would we be searching?

3.3.1 Defeating Attacks against Domain U Components

Attack attempting to prevent installation of the security monitor. We explain why such an attack cannot prevent the security monitor from being mapped into dom U memory.

- An attacker in the kernel cannot intercept and fake the hypercall that maps the 4MB memory into the kernel address space for dom U without being detected (then the system can be cleaned up before installing the protected monitor). Here the attacker deliberately does not actually make the real call to map the memory. However, this will be detected because call gate 1 will report failure because it identifies that the special 4M was never mapped.

- An attacker cannot interfere with the 4M mapping by calling Xen hypercalls themselves because of the following. (i) Our modifications to Xen ensure that attacker cannot map it before we do and cannot map only part of that memory. The latter is achieved because we store the shared memory's `mfn` in a page (step 4 of Figure 3.5), and after domain 0 grants the 4MB memory Xen will prevent domain U from using the hypercall `do_grant_table_op` to map the shared memory pages. Although the attacker could map 4MB using our call before we do, this just makes our mapping request redundant and does not cause any security problem because it's idempotent. (ii) Our modifications to Xen ensure that the attacker can neither unmap nor remap (any portion of) the 4MB after we map it. This is because after using our hypercall the `shared_memory` flag is changed to `mapped`, which prevents domain U remapping the shared memory, so that domain U cannot use our hypercall again to remap the shared memory to another virtual address. Moreover, using the hypercall `do_grant_table_op` cannot map or unmap part of that memory somewhere else.

- An attacker cannot fake the `malloc()` result, which is used in `ensure_shared_memory()`. Either malloc returns 2MB of allocated memory in the process address space or it doesn't. After the 1st call gate the hypercall we write will map it to M_2(readonly after the 2nd callgate) and M_3(readonly), then will protect the page table entries of these 2MB memory. So that attacker cannot map it to his own memory or write the M_3 to modify the signatures.

- An attacker that has compromised the kernel cannot modify the kernel's own page table in order to access the shared memory directly. Since the kernel can only modify page tables through Xen, even for the kernel's own page table, we can use Xen to prevent the kernel from modifying its own page table to access the shared memory. We use one bit in `page->u.inuse.type_info` (in Xen's `frame_table`) that we named `PGT_entry_protected` flag to mark the pages that need to protect. More specifically, there are three kinds of page table entries (PTE's) that need protection from modification:

1. Domain U kernel space mapping of M0-M1.
2. The domain U user application mapping of M2-M3 (M3 always readonly and M2 is only writable between the 1st callgate and 2nd callgate[6]).
3. The domain U user application has its own executable pages (these need to be protected to prevent TOCTOU attacks that change the executable after we first measure it, so that we just need to measure it during the 1st callgate).

The attacker (i.e., compromised kernel) has two ways to attack these three kinds of PTE's: First, the attacker could try to use his own page table entries to map to the M0-M3 or userapp executable pages, which seems possible because the kernel can set some page table entry with write access to it. However, the attacker cannot map his virtual address to M0-M3 in domain U, because these pages are owned by dom0, so that M0-M3's page table entries cannot be attacked by this way. And we marked the domain U user application executable pages as protected, so that the attacker who wants to map his own page table entries to user application will be detected by Xen which will then prevent this attack from succeeding. Second, the attacker can try to modify the PTE's of domain U's M0-M3 or the application's executable pages so as to let the PTE's map to the attacker's own pages. If the attack succeeds, the domain 0 may help the attacker to sign a wrong message. But we also prevent this kind of attack. We have mark these page table entries when the domain U map the shared memory, so that these page table entries cannot be modify by attacker.

Tampering with the protected monitor memory content. This is defeated because all reads, writes, and executes of bytes within the protected monitor's memory region are blocked by the hypervisor via the MMU. This means no software running within the VM can read, write, modify, or arbitrarily execute protected monitor code, irrespective of the CPU privilege level. Recall there is a special entry page ("jump page") that when executed deprotects the protected pages so that the PM can be invoked from outside the PM. The jump page contains only vectors (jumps) to specific known entry points, and cannot be read or written until execution in it is begun. As a result, the PM code and data cannot be tampered with in any way.

Starving the security monitor of the CPU. For this, the attacker would somehow prevent any user application from calling in to the PM. Because we are not attempting to entirely control the user VM, this attack must succeed against the prototype as we plan to implement it. Note this does not subvert the PM, nor guarantee access to resources controlled by the PM. It merely means the PM will not execute.

[6]Weiqi, are you sure M2 is only writable *between* the 1st callgate and 2nd callgate? Isn't it writable after the 2nd call gate also, until the 3rd call gate?

Regaining control of CPU when it is executing inside the security monitor. A major attack vector is to regain control of the CPU somehow while it is executing inside the protected monitor. The most obvious mechanism for this is scheduling a timer interrupt. We can take care of this by masking interrupts while inside the protected monitor. However, some system management interrupts (e.g., power events) are non-maskable and hence cannot be disabled by disabling interrupts. Thus there are a few intricate low-level attacks that the scheme is susceptible to. In particular, modifying BIOS or SMI code could be used to stage an attack [**?**]. Such attacks require considerable knowledge and skill and are frequently hardware-specific. Note that the attacker cannot regain control by causing VM faults, because ?????? [7]. If there were some way to regain control of the CPU while it was operating inside the monitor, there might be some way to use this to impersonate a user process and retrieve the key belonging to that process.

Impersonating the service caller. This is difficult[8] to do because the remote monitor inspects the binary making the invocation. So in order to impersonate the caller, the attacker must somehow either use the same binary or subvert the hashing process. In the first case, where the attacker somehow convinces the correct binary to disclose a secret, this is an attack against the application itself and is outside our security claim. We expect the second case, where the attacker subverts the hashing process to yield an incorrect result, to be quite difficult for the attacker. Since we perform hashing from outside domain U with the pages having already been forced into memory (preventing page fault handler attacks), the only way we can see to do this is to misrepresent which pages constitute the application in question, which is very difficult since we use the same data structures to determine the application pages as the CPU does when it executes them.

It's important to note that we hash the executable at the first call gate, and then lock the executable pages so they can't be modified. This is for two reasons: (i) The performance impact is lower, since there is a hash at the beginning instead of every time a message is sent. (ii) This prevents subtle TOCTOU attacks which would otherwise be possible (e.g., changing the binary just before sending the message, then somehow changing it back afterwards).

[7]Weiqi: Are the kernel VM mechanisms disabled here? Or would a VM fault eventually make its way to the domU kernel? How does Xen change the domU kernel VM fault process? Do we need to make sure that Xen won't pass a VM fault back to the domU kernel while we're in the protected monitor? Maybe it would be enough to make sure all of the pages of M0-M3 are in RAM?

[8]sx: a key attack here is: the attacker "clean" the user VM immediately before issuing the call — is the attack defeated? TPP: If you mean cleaning the entire contents of the VM, the attacker cannot do that because there is no way for the attacker to come back. If you just mean cleaning itself from the application code: Here we have to say that we cannot fix all security weaknesses in the application; that's out of our scope (see "first case" in the paragraph). If the application is designed in such a way that attacker can embed code in the data structures and then cause the application to execute this code (note for most applications this is a serious bug; very few applications need this quasi-self-modifying code type of trick. It would help to see the paper about the cleaning attack to double-check this and understand the attack better.(Also we can probably defeat this with the extension I suggested for contribution #1 if there was some way to make that work.)

3.3.2 Defeating Attacks against Domain 0 Components

Intuitively, the components in domain 0 cannot be attacked from domain U because domain 0 is inaccessible except via our communication mechanism. Nonetheless, we analyze possible attacks in more detail to ensure a correct analysis:

Attacks that attempt to penetrate domain 0. There are basically two ways an attacker could do this:

- Subvert the Xen hypervisor. This is very difficult to do from a guest domain (domain U) and is precluded by our assumptions. (Of course, the design of Xen is to preclude such attacks entirely.)
- Exploit some software the user has installed in domain 0 in order to control domain 0 from domain U. Here we must assume the user does not install some software in domain 0 that permits domain U to arbitrarily control, access, or modify domain 0. One way to achieve this for most users is to simply make domain 0 not be easily accessible.

Attacks against the domain 0 disk. The disk resides within the accessible space of domain 0. Domain 0 may choose to give domain U access to the disk, but without such explicit provision, domain U can see only the part of the disk that is designed for the use of domain U, if any. Generally hypervisors do not provide any access to the domain 0 disk by default, so our security here depends on the assumption that the hypervisor has not been configured to a configuration that allows domain U direct access to the domain 0 disk, and that no services have been installed in domain 0 that give domain U general access to the domain 0 disk. (Indeed, not allowing such access is a default and typical configuration for Xen, the hypervisor we chose).

Attacks that falsely request digital signatures. There's no way for an attacker in domain U to falsely request a digital signature from domain 0. This is because domain U falsely requesting digital signatures means either:

- pretending to be a different application or an uncompromised one (see Section 3.1).
- attacking the communication mechanism (see Section 3.3).

3.3.3 Defeating Non-Domain-Specific Attacks

Here we analyze attacks against components that are not contained within a specific domain.

Inter-domain communication. The attacker can't attack inter-domain communication from domain 0 because the attacker can't penetrate domain 0 (see the domain 0 analysis above). The attacker has the following options to attack inter-domain communication from domain U:

- **Attack kernel to block communication**

  This fails because the kernel is not involved in the communication process, because we deliberately designed our system so as to not use the kernel when communicating.

- **Attack application to block communication**

  - The attacker can't attack the application binary because it's protected by memory protection once communication is set up.
  - The attack can't attack memory pages with the communication data in them because they are protected from domain U access by anyone but the application (and the application can't be modified).
  - This leaves the possibility that the attacker can somehow disrupt communication by attacking internal data structures of the application in such a way as to disrupt communication. This depends on the quality of the implementation itself and is outside our scope – we don't attempt to protect the application itself from its own design and implementation.

- **Attack communication mechanism itself somehow**

  - **Attacker can't modify call gate**. The call gate is set up in the Global Descriptor Table (GDT), which by design in Xen can't be modified by domain U. [9]
  - **Attacker can't attack communication in application**. The cases and analysis from "attack application to block communication" above apply here, with the same result.
  - **Attacker can't attack communication in kernel**. As noted above, our design excludes the kernel from the communication process, so there is nothing here to attack.
  - **Attacker can't attack communication in hypervisor**. Attacking the Xen hypervisor from within a guest domain is very difficult as noted above and is precluded by our assumptions, which are standard.

Virtualization-based attacks. Some attacks use virtualization in some way to escalate an attacker to hypervisor privilege and hide a malware hypervisor from the operating system. Hardware virtualization technology attacks like Blue Pill [55] are not possible because they require executing virtualization instructions at ring 0 privilege, but Xen only allows domain U to run at ring 1 and higher. Similarly, SubVirt [39], which relies on adding a hypervisor early in the machine boot sequence, is not possible because the attacker is contained within domain U, and it can't support nested hypervisors anyway.

[9] Weiqi: this is correct, right?

3.4 Experimental Evaluation of Performance

3.4.1 Microbenchmark Performance of Inter-VM Communication

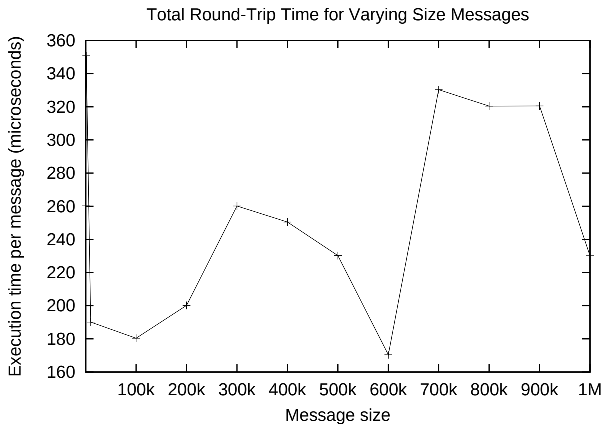


Figure 3.6: *Total* time required for message creation and processing for large round-trip messages. Merely sending the message both ways takes only 23 microseconds (OLD, T3400 machine). Until pause fixed, these numbers are pretty muchly meaningless because 1 minute 5 seconds becomes 15 seconds, 2m5s becomes 25 seconds, and 45 seconds becomes 45 seconds whereas the actual time is around 5 seconds. So the real numbers are somewhere between 100% and 11% of these numbers.

The time required to actually send and receive any message, including the two domain transitions that entails, is a mere 23 microseconds when we performed a simple performance experiment that sent 1 million messages (OLD, T3400 machine; needs to be redone). This included the time to hash the executable in domain U once. This assumes, however, that the client and server each want to send the same message to each other over and over (i.e, they only write the message to RAM once), and don't bother to read it. Thus we decided we should also create a microbenchmark where each side reads and writes the message it sends each time, because that time was more significant than the message send time.

Figure 3.6 shows total processing time required for simple large messages (i.e., the client and server read and write each message each time but do not perform any significant computation between reading and writing the messages). This includes the time to create a message in domU, send it to dom0, read it and write a reply message in domU, send it back to dom0, and read it in dom0. Time is measured from when the executable is invoked through when it sends 100(?) messages to when execution returns to the calling script. Each data point is averaged over 10(update to 20 when have busywait performance experiments) runs. Note that merely sending the message from dom0 to domU and back again requires only 23 microseconds; the bulk of the time is spent reading and writing the message in the buffer. The figure also includes the cost for hashing the domain U application once per invocation.

The smallest messages are 100 bytes, 1 kilobyte, and 10 kilobytes. The reader will note that for these small messages the message size has no visible influence on the message send time; only as messages increase from 10k to 100k does the time to actually access the memory begin to dominate the fixed per-message overhead. We believe this means that the effective speed of large messages is limited primarily by the memory bandwidth of the CPU and memory subsystem. Note that even a 1 megabyte message takes only a fraction of a millisecond to send.

3.4.2 Assured Signing Performance

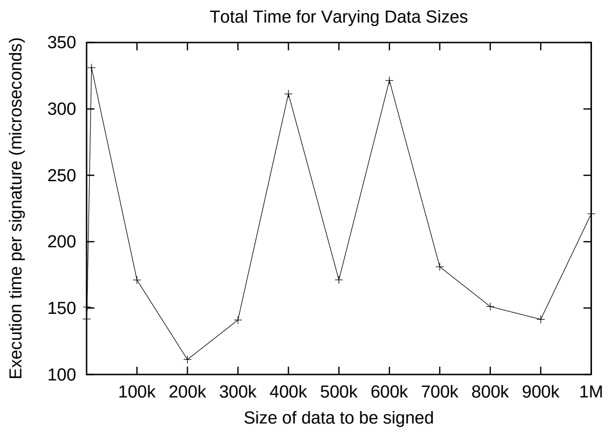


Figure 3.7: Time required to produce and verify signatures of varying sizes. Until pause fixed, these numbers are pretty muchly meaningless because 1 minute 5 seconds becomes 15 seconds, and 2m5s becomes 25 seconds, whereas the actual time is around 5 seconds. So the real numbers are somewhere between 100% and 20% of these numbers.

Figure 3.7 shows the performance of our system when creating signatures of varying sizes. Three curves are shown:

1. creating a signature directly in domain U by directly invoking the crypto library. This is insecure, since it provides no defense against attackers, and is shown only for comparison purposes.

2. creating a signature in domain 0 from a domain U request, which is our secure system, except with only limited remote verification, since we are using TPM quote values but not generating values for full TPM verification.

3. creating a signature in domain 0 from a domain U request, which is our full secure system, including generating values for TPM verification.

Note our prototype was designed primarily for simplicity, since it directly maps each call on the cryptlib API to a call to the secure domain. Coalescing these calls together using an intelligent communication layer would allow a significant reduction in the number of domain transitions (cf the domain transition and communication cost above, which is a fundamental cost in the PM design and while relatively low is still the most expensive new part of the system), which would significantly decrease execution time. Of course, simply redesigning the API could easily give an API that sends as little as one message. However, just redesigning the API would break transparency with existing clients.

As above, these performance numbers include hashing the client executable. In these tests the data to be signed does get copied once during the process, but this copy is required by the design of cryptlib. cryptlib takes a pointer to the data; its API provides no way to say where the data could be put to do zero copy processing – if it did then we could simply have the client place the data directly in the shared buffer.

3.5 Conclusion

We present an effective solution to malware attempts to compromise private signing keys or to falsely request digital signatures. Our solution not only completely secures the keys from the malware, but also can be used by existing applications without any modification to their source code. We also introduce a powerful mechanism for securely providing services to applications in a VM, which we believe will be of independent value. Finally, we demonstrate that our mechanisms have reasonable performance.

There are opportunities for future work. Most notably, we would like to determine how to measure the domain U kernel code, without interference from data structures and runtime patching that cause variation in the contents of the Linux 2.6 kernel code space. This would allow us to describe the state of the domain U kernel as part of our attested signatures, so that a verifier could attest that the kernel binary was not compromised. One way to do this would be to develop a comprehensive list of parts of the kernel that can change, and simply omit all of those when measuring. The challenge would be identifying these bytes in a way that is robust to changes in the kernel caused by continuing kernel development.

Misc Weiqi: [10]

[11]

[12]

[10] Weiqi ask:Can M_0 always be readonly? Domain U kernel seems never write it. I only write some things to M_1 like parameter and hash result is write in Xen space. Paul: Yes, M_0 can always be read-only for domain U (more secure even though prevents self-modifying code). If you are asking about domain 0, if you want it can be readonly for domain 0 after it is setup.

[11] Weiqi: We do need to enable the interrupt disablement code for call gate 2. Is that enabled?

[12] Weiqi: Please pay special attention to 2.2.4, 2.3.1, and all of Section 3. When I was correcting the English sometimes I couldn't tell what you meant to say, so I may have changed something wrong.

Chapter 4

Related Work

4.1 Introduction

Overview: In this chapter we organize the related work into two types: work related to the general goal of securing cryptographic secrets and processes, and work particularly related to the particular approaches we took. We discuss the general related work first, and then cover work that is related to only one particular piece in individual sections (Sections 4.6 and 4.7) following the general related work.

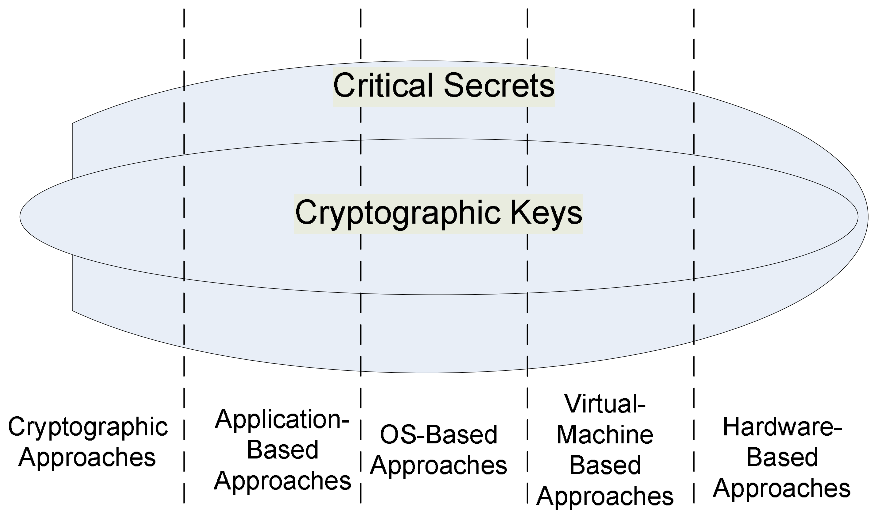

Critical Secrets
Cryptographic Keys
Cryptographic Approaches
Application-Based Approaches
OS-Based Approaches
Virtual-Machine Based Approaches
Hardware-Based Approaches

Figure 4.1: A categorization of the related work.

There is a broad set of related work. Figure 4.1 depicts the space of related work in a way that we hope is useful for the reader. The modified Venn diagram shows that some related work focuses on any kind of secrets and therefore applies to cryptographic keys as well. Some work applies only to cryptographic keys.

Related work can be characterized by the mechanism used to secure the critical secrets. The divided x axis in Figure 4.1 depicts that this space varies through hardware approaches (approaches rooted in trust in a specific hardware component), approaches based on trusting virtual-machine monitors, approaches based on trusting the operating system, approaches based on modifying the application, and lastly approaches based on cryptography. Certain types of cryptographically-based approaches only apply to the protection of cryptographic keys (but not other critical secrets), so the space of critical secrets that are also keys protrudes beyond the space of critical secrets that are not keys.

For simplicity, rather than categorizing related work into the ten categories implied in the figure, we characterize related work by the mechanism it uses to secure the critical secrets: hardware, virtual machines, conventional software, and cryptographic approaches that apply only to cryptographic keys. Often a solution will require modification to more than one of these levels, in which case we categorize it in to the leftmost level. E.g., if a solution requires both hardware and application changes, we categorize it as a hardware solution, partly because we expect hardware changes to be more difficult to deploy than software ones. A relevant recent survey can be found in **??**.

4.2 Protecting Secrets with Special Hardware

The most straightforward method to protect cryptographic keys and other secrets is to utilize some special hardware devices, such as cryptographic co-processors [74] or Trusted Platform Modules [30]. Still, such devices may be no panacea because they introduce hardware-related risks such as side-channel attacks [40]. Moreover, many systems do not have or support such devices.

4.2.1 Hardware Solutions: Trusted Platform Module

The vast majority of hardware solutions proposed for securing critical secrets rely on the Trusted Platform Module, or TPM, proposed by the Trusted Computing Group [30].

We note that our work has several points of superiority compared to a typical TPM-based system:

- Our system does not require special hardware, unlike TPM, although we can leverage a TPM to provide additional assurance to a remote verifier of signatures.
- We provide better performance, partly because the TPM is frequently handicapped by its LPC bus, which was required to avoid too much cost.
- Our system can also be upgraded, whereas the TPM design deliberately precludes upgrades.

- Our protected monitor platform (Chapter 3) does not fundamentally depend on the integrity of the kernel, whereas functionality of the TPM software stack does depend on kernel integrity (for example, it depends on the TPM device driver)
- Our platform's capabilities are much more general than what the TPM directly supports. For example, our protected monitor can execute arbitrary code, including calling into the operating system kernel.
- Significantly, we believe we are not subject to various kinds of binary-replacement attacks that apply to typical software checksumming (note the TPM has to rely on software to compute the hash of a binary–due to its low-bandwidth bus it could not perform hardware-checksumming even if it were part of the design). Oorschot capably lays out several such attacks in [68] and [64]. Essentially, these attacks defeat "self-hashing" code by utilizing "operating system level manipulation of processor memory management hardware" on compromised kernels. Since the hypervisor is at a higher level of abstraction (and in fact is often responsible for managing the illusion of direct access to that memory management hardware), it is not subject to such attacks. In fact, our external verifier is essentially completely isolated by VM isolation.

Caveat: This imperviousness to some checksumming attacks does not come entirely for free; virtual-machine introspection has to rely on kernel-level data structures in the VM in order to establish the pages that consitute the code for a given process, for example. The technical report [61] studies implications of the reliance of virtual-machine introspection tools on the integrity of kernel data structures, concluding that efficacy of VM-based introspection typically still relies on data structures the VM can manipulate, and gives examples of attacks. "Our attacks undetectably hide a kernel module, hide a running process, and add Trojan versions of critical software." [1] However, they also develop a tool that can still perform some monitoring without being subject to such attacks.

We also provide an explicitly-secured CSP, and can provide secure auditing for its operation. This prevents a cryptographic-service-provider imposter attack where encryption isn't done, and also prevents compromised encryption routines from making a copy of the data . Because our logs are stored in an inaccessible protection domain, our logs cannot be tampered with or destroyed from the insecure domain, greatly reducing vulnerability to denial-of-service attacks on logging.

[**?**] demonstrates a Time-Of-Check Time-of-Use attack on a TPM system. The application binary is modified after the TPM computes the hash but before the binary is executed.

[1] Although we make no attempt to demonstrate it, we believe such attacks can be defeated, at least in-principle. A powerful way to defeat these attacks would be by actually running the code in question from the hypervisor, since when the code is running it must provide the actual version of itself to be executed.

4.2.2 Protecting Functions with TPM

More recent versions of the TPM support a new functionality mode that allows the launch of highly-isolated signed code, when used with a CPU with appropriate support (Intel's Trusted Execution Technology (TXT), or AMD's Secure Virtual Machine technology (SVM), which are included in many of their recent CPU's). This allows a small piece of Secure Loader Block (SLB) code to launch in a completely protected environment, including disabling all other CPU cores and typically DMA as well. Unfortunately, aside from the special hardware, this suffers from a number of limitations. Only 64k of code can be executed at a time in this fashion. This code cannot have any dependencies on other software in the system, e.g., it cannot call into other pieces of code. Invocation of the SLB code is frequently too slow to use for many purposes [47], and moreover the there is the impact on system performance of disabling all other CPU's, CPU cores, and threads of CPU execution (e.g., Hyperthreading). Because of the slowness and the difficulty of interacting with any other code in the system, the TXT/SVM mechanism is not suitable for hooking into the kernel.

Flicker [46] builds on the TXT/SVM technologies, greatly simplifying the development of SLB code for an application and providing additional useful functionality like secured storage between executions of the SLB code. However, in the end it cannot overcome the fundamental limitations of the technology as designed and implemented in the TPM and CPU hardware. In particular, even though Flicker could be used to check for the existence of hooks in a kernel, it could not be used to service those hooks because SLB invocation is too slow.

The Terra paper [26] builds an impressive edifice on a machine with a secure coprocessor similar to a TPM. Virtual machines run within a Trusted Virtual Machine Monitor, as one of two types. Open-box VM's can run any operating systems and software. Closed-box VM's run only software stacks attested by the TVMM (the entire stack must be attested). Measuring an entire VM requires an extremely large number of combinations be the same as certified. Moreover, there is no facility for securely examining or controlling what's going on within a VM; closed-box VM's are entirely independent of open-box VM's that users could run their own choice of software within. Although it is not emphasized, Terra appears to assume the entire hardware platform is tamper-resistant , not merely the trusted co-processor.

4.2.3 Other Hardware Solutions

The most straightforward hardware method to protect cryptographic keys is to utilize a special hardware device for cryptographic processing such as a cryptographic co-processor [74]. Still, such devices may be no panacea because they introduce hardware-related risks such as side-channel attacks [40]. Moreover, many systems do not have or support such devices.

"Architecture for Protecting Critical Secrets in Microprocessors" [42] proposes an elaborate and thorough "secret-protected" hardware architecture to protect against software and DMA attacks. The work is impressive and complete, with features such as cryptographic keys that follow their users between devices, rather than being tied to particular devices. However, it is highly-complex in addition to requiring changes to the CPU and operating system, and we suspect is thus unlikely to be used in practice.

[60] proposes managing security at the level of memory regions rather than only at the level of processes, giving a finer level of granularity and simplying shared access to secret data in memory. It proposes small CPU hardware changes to make this more efficient, such as having a hardware cache in the CPU for the memory access descriptors, and uses encryption for confidentiality of data, code, and security descriptors.

[24] assumes that computers will largely adopt non-volatile RAM due to potential advantages such as lower power consumption and "instant on" starts. This leads to a new security risk: adversaries reading the RAM of a powered-down system. The authors solve this problem by introducing a small Memory Encryption Control Unit, or MECU, between the CPU cache and RAM, so that all data stored in actual RAM will be encrypted. Using AES to generate a one-time pad while the memory fetch is ongoing and then simply XOR'ing with the pad allows the performance hit of encryption to be minimal. However, the pad has to generate substantial amounts of key material with a low latency in order to keep up with the substantial memory bandwidth of modern CPU's and DMA devices such as graphics cards, so we believe that even in quantity MECU chips could not be cheap. Additional complexity, or substantial performance hits, come from maintaining coherency in the tables between the multiple MECU's on a system.

InfoShield [59] enforces "information usage safety" as described in program semantics by extending hardware with secure load and store operations and encrypting sensitive data when it is stored to memory. Infoshield relies on the semantics of the original source code to be correct, and requires annotation to specify which data is sensitive.

4.3 Protecting Secrets and Functions with Virtual Machines

Many recent works utilize virtual machines to help secure critical secrets. Perhaps the most general and relevant of these are the works that use VMM's to encrypt application pages for confidentiality against any other accessor, including the running operating system. Overshadow [18] does this with "multi-shadowing", where a VMM can present the illusion of multiple versions of a page of physical RAM to a client VM. This allows an application to quickly access unencrypted versions of a page while ensuring the OS and any other processes see only the encrypted version of the page. This also encrypts files on disk, because the data on the page is already encrypted when the OS accesses it for a disk transfer. This requires modifications to the VMM, in this case VMware Workstation, as well as a shim that runs when applications first load, but means that the applications themselves and Linux kernel can run unmodified. Although technically they do not modify the OS, they do require applications to use a special loader and shim runtime. We do not modify the OS nor applications at all, although we do hook into the OS. Whether their approach could actually be used in Windows is not clear, since it doesn't easily view resources as memory pages.

Since Overshadow is one of the most closely-related non-hardware solutions to our work, we examine some additional points of comparison. We believe that our solution is rather more flexible than Overshadow. For example, Overshadow's design appears to require that protection domains be completely isolated from one another; there is no provision for protecting information other than enclosing it within a protection (encryption/integrity) domain. So if an application needs to be able to access secured data files belonging to another application, the two applications must be in the same protection domain. By contrast, we not only can allow multiple applications to access the same protected data if desired, but we support policies which can be used to specify in detail what data files are shared and how. It is not clear to us whether Overshadow requires all data on the system to be within some protection domain; if so, we speculate that many existing applications would be difficult to use without putting all of them in the same protection domain, which would greatly reduce the security added. Additionally, this would mean Overshadow does not allow even the sharing of unprotected data, since there would be no unprotected data.

The performance impact of Overshadow can be substantial, because of the CPU impact of decrypting or encrypting a page whenever access alternates between the application and the operating system. This is more visible in some contexts than others. For example, a UNIX `fork` microbenchmark performs at only 20% of native performance without Overshadow. Actual applications performed no slower than 80% of native performance when only anonymous pages were encrypted. When all pages and files were encrypted, performance was lower; in particular Apache's throughput was less than 50% of its throughput compared to running without Overshadow. We do not believe our CPU impact and total performance impact will be as significant.

Additionally, Overshadow provides only moderate protection against physical RAM disclosure attacks, because pages in physical RAM are encrypted only if the page's last accessor was the operating system. However we expect such attacks to be difficult for malware compared to attacks disclosing the virtual memory space. Overshadow might foil attacks against the virtual memory space, depending on who accessed the pages last and whether the attack comes through the kernel, which would cause Overshadow the encrypt the pages.

[73] is a similar work published concurrently that uses a similar technique on the Xen hypervisor, using manipulation of the VM's TLB to provide access to encrypted and unencrypted versions of page frames.

We expect this work would compare much the same against ours as Overshadow.

4.3.1 Sujit Sanjeev Master's Thesis

Sujit Sanjeev first implemented the concept of a cryptographic service provider secured by a VMM, as detailed in the master's thesis [56]. We conceived the same idea independently, but his implementation was complete when ours was still being planned. Since our signature solution provider is based on a partial cryptographic service provider, we note a few of the important differentiations of our work:

1. We use the virtual machine monitor Xen, which has excellent performance and is suitable for production use. Their work uses lguest, a minimal hypervisor designed for ease of implementation and modification, where performance suffers because the chief aim is simple code. Lguest is a simple kernel module that provides multiple virtual machines on the same kernel by multiplexing kernel data structures, using the kernel's existing paravirtualization support for privileged operations. Not only does this mean all virtual machines are running the same version of the kernel (because they're running the same kernel code), but it appears to also have security implications.

2. We use a production-grade cryptographic implementation, which would be suitable for actual use in practice. Their work relies on the Linux kernel cryptography implementation, which is designed only to suffice for expected kernel use, such as IPSEC and dm-crypt.

3. Their work contains no provision for key management. Indeed, it appears that only a single key can be used, and may even be hardwired into the code.

4. Because their cryptographic service provider is running in the hypervisor, it has certain limitations. In particular, there is no facility for persisting data, so keys cannot be stored by the provider, which would be more secure; instead they have to be stored in the user VM.

We know of no plans to publish the work in [56] beyond the master's thesis.

4.3.2 Other Virtual Machine Related Work

The technique of *virtual machine introspection* [27] examines the contents of a virtual machine from outside the VM. Compared to our protected monitor foundation, typical virtual machine introspection has the following disadvantages:

1. The *semantic gap* problem: the virtual machine state is much more easily interpreted from inside the VM's context than from outside. In other words, it's very difficult to piece together what's going on inside the VM from outside.

2. Introspection cannot be used to hook functions, because it provides only the ability to examine the state of the VM.

[41] uses virtual machines to isolate the use of critical secrets from the user's ordinary operating system. Whenever a user needs to use a critical secret for authenticating themselves, they use a special non-interceptable UI command (e.g., CTRL-ALT-Delete) to switch to the VMM and then switch to a secure VM. The critical secret is input there and appropriately transmitted, e.g., to a remote Web site that explicitly requests it from the secure VM. The secure VM relays the authentication success to the ordinary VM when switching back to it. Unfortunately, this means the user has to learn new behavior and the client software and server software both have to be modified to support Vault.

4.4 Protecting Secrets via Conventional Software

4.4.1 Protecting Keys

We begin by examining approaches to enhance the secrecy of cryptographic keys against attacks that may exploit system vulnerabilities. Here we elaborate the basic ideas of investigations under this approach, assuming that no copies of a key appear in unallocated memory (see [20, 32] for examples of techniques that address this issue). Later in this section we will examine in detail certain work on critical secrets that is particularly closely related to our work. Without loss of generality, suppose a cryptographic key is stored on a hard drive (or memory stick), fetched to RAM to use, and occasionally swapped to disk. Thus, we consider three aspects.

- Safekeeping cryptographic keys on disk: Simply storing cryptographic keys on hard drives is not a good solution. Once an attacker has access to the disk (even the raw disk) the key can be compromised through means such as an entropy-based method [57]. The usual defense is to use a password to encrypt a cryptographic key while on disk. However, an attacker can launch an off-line dictionary attack against the password (Hoover and Kausik [34] is an exception but with limitations). A more sophisticated protection is to ensure "zero" key appearances on disk (i.e., a key never appears in its entirety on disk). For example, Canetti et al. [14] exploit an all-or-nothing transformation to ensure an attacker who has compromised most of the transformed key bits still cannot recover the key.

- Safekeeping cryptographic keys when swapped to disk: The concept of virtual memory means that cryptographic keys in RAM may be swapped to disk. Provos [54] presents a method to encrypt swapfile for processes with confidential data. (In a different setting, Broadwell et al. [11] investigate how to ship crash dumps to developers without revealing users' sensitive data.)

- Safekeeping cryptographic keys in RAM: Ensuring secrecy of cryptographic keys in RAM turns out to be a difficult problem, even if the adversary may be able to disclose only a portion of RAM. Recent investigations by Chow et al. [19, 20] show some best practices in developing secure software (e.g., clearing sensitive data such as cryptographic keys promptly after their use, stated years ago by Viega et al. [65, 66]) have not been widely or effectively enforced. Moreover, Harrison and Xu [32] found that a key may have many copies appearing in RAM. The present work makes a significant step beyond [32] by ensuring there are no copies of the key appearing in RAM. As a side product, our Key-in-Register method in Chapter 2 should defeat the impressive recent attack of extracting cryptographic keys from DRAM chips when the computers are inactive or even powered off [?] because a because a key never appears in its entirety in RAM. This work also highlights that it may be necessary to treat RAM as untrusted, per our work.

4.4.2 Microsoft Windows Key Protection

As an example of common practice we look at Microsoft Windows. Windows standards provide for the use of cryptography via a Cryptographic Service Provider [49], such as the one bundled with Windows, and more recently the Cryptography API: Next Generation (CNG) [48]. It appears that long-lived private keys are supposed to be isolated from application processes (and hence presumably should not appear in process RAM) as of Windows Vista and Windows Server 2008, but not in earlier versions of Windows [50]. (Windows XP Service Pack 3 does include `fips.sys`, a kernel-mode cryptographic module compliant with FIPS 140-1 Level 1, which can provide services to other kernel mode drivers. We found no reason to believe these operations are made available to user-land applications.)

4.4.3 Protecting General Secrets

XFI [63] is a pure software mechanism that uses a binary rewriting with a binary verifier to enforce fine-grained memory access control. This provides access control for critical secrets when stored in RAM, as long as all programs have had their binaries rewritten and verified. [12] proposes adding small CPU hardware changes to increase the efficiency of XFI, as well as the efficiency of a related mechanism that enforces control-flow integrity in order to make it more difficult to hijack program control flow. Tightlip [75] takes an interesting approach to securing user secrets; when unauthorized applications access files containing user secrets, a “doppelganger” duplicate process is created, which gets a sanitized version of the bytes from the file. The doppelganger and the original process run in parallel, until one attempts to communicate some output that is different from the other, at which time a privacy breach might be occurring, and so a policy decision must be made, e.g., whether to replace the original with the doppelganger or to allow the output of the original.

4.5 Protecting Keys Cryptographically

A completely different approach to protecting cryptographic keys is to mitigate the damage caused by their compromise. Notable results include the notions of threshold cryptosystems [22], proactive cryptosystems [51], forward-secure cryptosystems [3, 7, 8], key-insulated cryptosystems [23], intrusion-resilient cryptosystems [36]. [70] proposes a model for understanding digital signature security of credential infrastructures in the presence of key compromise and proposes engineering techniques to improve it.

Another approach to protecting cryptography against memory disclosure attacks is taken by [1], which shows that certain cryptosystems are naturally resistant to partial-key-exposure memory disclosure attacks, in the sense that a large fraction of the key bits can be disclosed without endangering the secrecy of the actual key. Nevertheless, our experience shows that it may be likely that memory disclosure attacks, once successful, will expose a cryptographic key in its entirety when no countermeasures like those presented in this work are taken.

All of these techniques are orthogonal to our approach, and hence may be combined with our work.

4.6 Work Specifically Related to Securing Signatures

Digital signing is one of the most important cryptographic tools for security because it can be used to enforce/ensure integrity, authentication, non-repudiation, and data provenance (which is an emerging application for evaluating the trustworthiness of data items [?, ?, 69]). The state of the art is that these properties of digital signature can be rigorously proven assuming (in additional certain complexity-theoretically hard problems) that the private signing keys are kept absolute secret (in the black-box model) or that the leaked information about the private signing keys (because of side-channel attacks) is upper bounded. However, there is another class of attacks, which can be launched by malicious malware, that aim to compromise (or steal) the private signing keys in their entirety (rather than partial information about them) or to compromise the private signing functions (without stealing the keys). While there have been some studies addressing this issue, many problems remains open and this paper moves a significant step towards solving the problem. Specifically, we aim to enhance the assurance of digital signatures even if the attacker can penetrate into victim systems to steal the private signing keys or compromise the private signing functions. We present the design, implementation, and evaluation of a light-weight system, which takes advantage of both trusted computing and virtualization simultaneously. The core of the system is a novel technique we call *protected monitor*, which can retrofit security applications on user desktops and in cloud computing in a transparent fashion (i.e., without requiring modification to the operating system or applications) and thus might be of independent value.

Recently, Xu and Yung [70] classified attacks against the key into two families: (i) hit-and-run, where the attacker penetrates into a system, steals the private signing keys, and leaves the victim computer (while erasing the traces of compromise); (ii) hit-and-stick, where the attacker penetrates into a system and resides on the victim computer with or without stealing the private signing key. A hit-and-run attacker can compromise private signing keys in their entirety (rather than side-channel attacks caused partial information leakage [1]). A hit-and-stick attacker can compromise the private signing functions, even if the private signing keys are not compromised. A hit-and-stick attack is much more powerful, as long as the attack is kept stealthy, because the attacker can essentially get digital signatures on whatever messages the attacker wants to sign. A novel approach to defeating hit-and-run attacks was presented in [70], while the problem of defeating hit-and-stick is left open. In Chapter 3 we presented an effective solution to this problem.

Note the problem of securing signatures cannot be solved by simply deploying tamper-resistant hardware because a malicious piece of malware, which has penetrated into the OS to which the hardware device is attached, can likely issue any legitimate instructions to the hardware. As a consequence, even if the private signing keys are not compromised (because it never leaves the tramper-resistant hardware devices), the corresponding signing functions are compromised (because the attacker can get real digital signatures anyway).

4.7 Work Specifically Related to the Protected Monitor

VM introspection has become an important security mechanism. The initial idea [17, 25] was to exploit hypervisors for isolating intrusion detection systems (IDS) from the systems they monitor, but was later extended by numerous studies. For example, one can insert traps into the monitored VM so as to capture certain events [4], where the monitor code executes either in the hypervisor or in a trusted VM. This is different from our protected monitor because our security monitor resides directly in the User VM, meaning it has more power to bridge the semantic gap in VM introspection (e.g., our security monitor could understand the semantics of objects like the kernel).

In many ways the work that is most related to our protected monitor is Sharif et al.'s secure in-VM monitoring [58], which takes advantage of hardware-supported virtualization to achieve better introspection.

That work only performs virtual machine introspection and monitoring of the untrusted VM; no provision is made for secure communication between applications and the secure VM. This could be emulated to a limited extent by having the secure VM examine the untrusted VM and try to read application data, but there is no mechanism for it to communicate data back to applications in the untrusted VM, and it also does not allow for synchronous function invocation (applications would need to use something like a shared-memory busywait model). There is no memory protection of the application data and no protection of the application or the communication process from the kernel or other applications. Moreover, their work requires Intel's hardware support for virtualization (Virtualization Technology, or VT), limiting them to recent Intel CPU's (presumably their work could be ported to AMD's similar mechanism), whereas Xen can run on essentially any Intel-compatible CPU (we need only 386 and higher with PAE support, which was introduced in the mid 1990's).

Lares [52] extends VM introspection by providing using Xen's memory protection to protect hooks placed inside the guest kernel, including placing a small piece of "trampoline" code inside the guest VM where the hooks go to in order to communicate back to the secure VM. There is no functionality for hooking user-land applications nor for communicating with them.

Chapter 5

Conclusion

5.1 Summary

This proposal sets forth three pieces:

1. *Safekeeping Cryptographic Keys from Memory Disclosure Attacks* (Chapter 2) is a technique for using a cryptographic key without ever having the key in memory. This gives protection against memory disclosure attacks which otherwise can frequently recover keys, particularly in the case of Apache on Linux [32]. As a specific example, a prototype is created that modifies RSA private key encryption in OpenSSL to use the technique.

   The key point is that we can completely protect keys from memory disclosure attacks, even hardware ones such as Firewire ([**?**]) while requiring no special hardware (only resources found in typical CPU's). Because we prototype this on a single-core machine, we have to use a RAM scrambling technique to store the key in the single-CPU-core case, so we show that common attacks such as entropy scanning, signature scanning, and content scanning are infeasible.

2. The *Protected Monitor* (Chapter 3) serves as a foundation for the third piece, and could also be useful for many other security applications, because it provides a platform on which secured services can be built. It is particularly well-suited to securing against malware attacks, although it can be used for many other types. The monitor's architecture gains memoru protection from a virtual machine manager but still allows the monitor to operate from within the memory space of the virtual machine, unlike virtual machine introspection. This secures the monitor against most attacks from the user VM while still allowing services built on the platform to interact with the kernel.

3. The *Secure Signature Service Provider* (Chapter 3) allows clients of digital signatures to have high-confidence and remotely attestable secure digital secures and key storage, even in the presence of malware running at elevated privilege levels. Key storage services are secure against malware and even raw disk access (from within the VM). Callers are heavily validated and the secure domain can be attested by the TPM if desired so that remote verifiers can have high confidence in the authenticity of the signatures. Moreover, the design provides for a smaller TCB for cryptographic operations, since the cryptography implementation can rely on a smaller and controlled software stack.

5.2 Future Work

There are several opportunities for useful future work mentioned in the chapters referenced above. There are also two major components that we believe would be especially useful to build on the protected monitor and plan as future work:

- The *VM-Isolated Cryptographic Service Provider* would allow clients of cryptographic services to have high-confidence cryptography and key storage, even in the presence of malware running at elevated privilege levels. Basically this can be done by extending the crypto implementation used for the signature service provider into a general crypto service provider. We will use a flexible policy mechanism to express what applications may use what keys and cryptographic services in terms of rules describing various criteria including suspicious malware behavior. As with the signature service provider, key storage services are secure against malware and even raw disk access (from within the VM). Callers are heavily validated (authentication, provenance-checking, and checking for malware behaviors that may indicate the calling application is infected with malware). Moreover, the design provides for a smaller TCB for cryptographic operations, since the cryptography implementation can rely on a smaller and controlled software stack.

- *Transparent Critical Secrets Protection* would transparently secure critical secrets on disk from disclosure via malware (such as for identity theft). No modifications would be required for legacy applications nor for the operating system. The persistent storage is not accessible without authentication and approval, even with raw disk access (from within the virtual machine). The goal is to have files with secrets are identified automatically; the user does not have to manually specify files or policies. The user may specify policies if desired.

Bibliography

[1] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In Omer Reingold, editor, *TCC*, volume 5444 of *Lecture Notes in Computer Science*, pages 474–495. Springer, 2009.

[2] AMD. Amd64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, May 2005.

[3] R. Anderson. On the forward security of digital signatures. Technical report, 1997.

[4] K. Asrigo, L. Litty, and D. Lie. Using vmm-based sensors to monitor honeypots. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06)*, pages 13–23, 2006.

[5] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *ACM Conference on Computer and Communications Security*, 2010.

[6] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO'01*, pages 1–18, 2001.

[7] M. Bellare and S. Miner. A forward-secure digital signature scheme. In M. Wiener, editor, *Proc. Crypto'99*, pages 431–448. Springer-Verlag, 1999. Lecture Notes in Computer Science No. 1666.

[8] M. Bellare and B. Yee. Forward-security in private-key cryptography. In *Cryptographer's Track - RSA Conference (CT-RSA)*, pages 1–18. Springer-Verlag, 2003. Lecture Notes in Computer Science No. 2612.

[9] Eli Biham. A fast new des implementation in software. pages 260–272. Springer-Verlag, 1997.

[10] Boneh, Durfee, and Frankel. An attack on RSA given a small fraction of the private key bits. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*. LNCS, Springer-Verlag, 1998.

[11] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proceedings of Usenix Security Symposium 2003*, pages 273–284, 2004.

[12] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural support for software-based protection. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 42–51, New York, NY, USA, 2006. ACM.

[13] E. Bursztein, S. Bethard, C. Fabry, J. Mitchell, and D. Jurafsky. How good are humans at solving captchas? a large scale evaluation. In *IEEE Symposium on Security and Privacy*, pages 399–413, 2010.

[14] R. Canetti, Y. Dodis, S. Halevi, E. Kushilevitz, and A. Sahai. Exposure-resilient functions and all-or-nothing transforms. In *EUROCRYPT*, pages 453–469, 2000.

[15] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *ACM Conference on Computer and Communications Security*, pages 555–565, 2009.

[16] D. Chaum and E. Van Heyst. Group signatures. In D. W. Davies, editor, *Advances in Cryptology — Eurocrypt '91*, pages 257–265, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science No. 547.

[17] P. Chen and B. Noble. When virtual is better than real. In *HotOS*, pages 133–138, 2001.

[18] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2008. ACM.

[19] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of Usenix Security Symposium 2004*, pages 321–336, 2004.

[20] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime. In *Proc. 14th USENIX Security Symposium*, August 2005.

[21] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. Intel Corporation, 2007.

[22] Y. Desmedt and Y. Frankel. Threshold cryptosystems. pages 307–315. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[23] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 65–82. Springer, 2002.

[24] William Enck, Kevin R. B. Butler, Thomas Richardson, Patrick McDaniel, , and Adam Smith. Defending against attacks on main memory persistence. Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC),December 2008. Anaheim, CA. FIXME WITH CCSB WHEN AVAILABLE.

[25] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'03)*, 2003.

[26] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, volume 37, 5 of *Operating Systems Review*, pages 193–206, New York, October 19–22 2003. ACM Press.

[27] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[28] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[29] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308, April 1988.

[30] Trusted Computing Group. https://www.trustedcomputinggroup.org/.

[31] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys, August 2008.

[32] K. Harrison and S. Xu. Protecting cryptographic keys from memory disclosure attacks. In *IEEE DSN'07*, pages 137–143, 2007.

[33] K. Harrison and S. Xu. Protecting cryptographic keys from memory disclosures. In *Proceedings of the 2007 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS'07)*, pages 137–143. IEEE Computer Society, 2007.

[34] D. Hoover and B. Kausik. Software smart cards via cryptographic camouflage. In *IEEE Symposium on Security and Privacy*, pages 208–215, 1999.

[35] Intel. Intel trusted execution technology mle developers guide. `http://www.intel.com/technology/security/`, June 2008.

[36] G. Itkis and L. Reyzin. Sibir: Signer-base intrusion-resilient signatures. volume 2442 of *Lecture Notes in Computer Science*, pages 499–514. Springer-Verlag, 2002.

[37] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 2005 Symposium on Operating Systems Principles (SOSP'05)*, pages ???–???, 2005.

[38] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09)*, pages 115–124, 2009.

[39] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 2006. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[40] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. pages 104–113. Springer-Verlag, 1996. Lecture Notes in Computer Science No. 1109.

[41] Peter C. S. Kwan and Glenn Durfee. Practical uses of virtual machines for protection of sensitive user data. In Ed Dawson and Duncan S. Wong, editors, *Information Security Practice and Experience, Third International Conference, ISPEC 2007, Hong Kong, China, May 7-9, 2007, Proceedings*, volume 4464 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2007.

[42] Ruby B. Lee, Peter C. S. Kwan, John Patrick McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA 2005)*, pages 2–13, 2005.

[43] P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, S. Turner, and J. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proc. 21st National Information Systems Security Conference (NISSC'98)*, 1998.

[44] P. Loscocco, P. Wilson, J. Pendergrass, and D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing (STC'07)*, pages 21–29, 2007.

[45] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys'08)*, 2008.

[46] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. In Joseph S. Sventek and Steven Hand, editors, *EuroSys*, pages 315–328. ACM, 2008.

[47] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How low can you go?: recommendations for hardware-supported minimal TCB code execution. In Susan J. Eggers and James R. Larus, editors, *ASPLOS*, pages 14–25. ACM, 2008.

[48] Microsoft Developer Network. Cryptography api: Next generation (windows). http://msdn2.microsoft.com/en-us/library/aa376210.aspx, September 2007.

[49] Microsoft Developer Network. Cryptography (windows). http://msdn2.microsoft.com/en-us/library/aa380255.aspx, October 2007.

[50] Microsoft Developer Network. Key storage and retrival (windows). http://msdn2.microsoft.com/en-us/library/bb204778.aspx, September 2007.

[51] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 51–59. ACM Press, 1991.

[52] Bryan D. Payne, Martim Carbone, Monirul I. Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, pages 233–247. IEEE Computer Society, 2008.

[53] D. Piegdon and L. Pimenidis. Hacking in physically adressable memory. In *Proc. 4th International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'07)*, 2007.

[54] N. Provos. Encrypting virtual memory. In *Proceedings of Usenix Security Symposium 2000*, 2000.

[55] Joanna Rutkowska. Subverting vista kernel for fun and profit. Black Hat Briefings, Las Vegas, August 2006.

[56] Sujit Sanjeev. Protecting cryptographic keys in primary memory using virtualization. Unpublished master's thesis, Arizona State University, 2008.

[57] Shamir and van Someren. Playing 'hide and seek' with stored keys. In *FC: International Conference on Financial Cryptography*. LNCS, Springer-Verlag, 1999.

[58] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *ACM Conference on Computer and Communications Security (CCS'09)*, pages 477–487, 2009.

[59] W. Shi, J. B. Fryman, G. Gu, H. H. S. Lee, Y. Zhang, and J. Yang. Infoshield: a security architecture for protecting information usage in memory. *2006. The Twelfth International Symposium on High-Performance Computer Architecture*, pages 222–231, February 2006.

[60] Weidong Shi, Chenghuai Lu, and Hsien-Hsin S. Lee. Memory-centric security architecture. In Thomas M. Conte, Nacho Navarro, Wen mei W. Hwu, Mateo Valero, and Theo Ungerer, editors, *HiPEAC*, volume 3793 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2005.

[61] Abhinav Srivastava, Kapil Singh, and Jonathon Giffin. Secure observation of kernel behavior. Unpublished technical report: http://www.cc.gatech.edu/research/reports/GT-CS-08-01.

[62] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems (EuroSys'10)*, pages 209–222, 2010.

[63] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: software guards for system address spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[64] Paul C. van Oorschot, Anil Somayaji, and Glenn Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Trans. Dependable Sec. Comput*, 2(2):82–92, 2005.

[65] J. Viega. Protecting sensitive data in memory. http://www.cgisecurity.com/lib/protecting-sensitive-data.html, 2001.

[66] J. Viega and G. McGraw. *Building Secure Software*. Addison Wesley, 2002.

[67] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy*, pages 380–395, 2010.

[68] Glenn Wurster, Paul C. van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *IEEE Symposium on Security and Privacy*, pages 127–138. IEEE Computer Society, 2005.

[69] S. Xu, H. Qian, F. Wang, Z. Zhan, E. Bertino, and R. Sandhu. Trustworthy information: Concepts and mechanisms. In *Proceedings of 11th International Conference Web-Age Information Management (WAIM'10)*, pages 398–404, 2010.

[70] S. Xu and M. Yung. Expecting the unexpected: Towards robust credential infrastructure. In *Financial Cryptography*, 2009.

[71] Shouhuai Xu and Moti Yung. Expecting the Unexpected: Towards Robust Credential Infrastructure. In *2009 International Conference on Financial Cryptography and Data Security (FC'09).*, February 2009.

[72] W. Xu, G. Ahn, H. Hu, X. Zhang, and J. Seifert. Dr@ft: Efficient remote attestation framework for dynamic systems. In *Proceedings of 15th European Symposium on Research in Computer Security (ESORICS'10)*, pages 182–198, 2010.

[73] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In David Gregg, Vikram S. Adve, and Brian N. Bershad, editors, *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008*, pages 71–80. ACM, 2008.

[74] B. Yee. *Using secure coprocessors*. PhD thesis, Carnegie Mellon University, May 1994. CMU-CS-94-149.

[75] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *NSDI*. USENIX, 2007.

[76] Jing Zhang, Adriane Chapman, and Kristen LeFevre. Do you know where your data's been? - tamper-evident database provenance. In Willem Jonker and Milan Petkovic, editors, *Secure Data Management*, volume 5776 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2009.