Enhancing Data Trustworthiness via Assured Digital Signing

Weiqi Dai, T. Paul Parker, Hai Jin, and Shouhuai Xu

Abstract—Digital signatures are an important mechanism for ensuring data trustworthiness via source authenticity, integrity, and source nonrepudiation. However, their trustworthiness guarantee can be subverted in the real world by sophisticated attacks, which can obtain cryptographically legitimate digital signatures without actually compromising the private signing key. This problem cannot be adequately addressed by a purely cryptographic approach, by the revocation mechanism of Public Key Infrastructure (PKI) because it may take a long time to detect the compromise, or by using tamper-resistant hardware because the attacker does not need to compromise the hardware. This problem will become increasingly more important and evident because of stealthy malware (or Advanced Persistent Threats). In this paper, we propose a novel solution, dubbed *Assured Digital Signing* (ADS), to enhancing the data trustworthiness vouched by digital signatures. In order to minimize the modifications to the Trusted Computing Base (TCB), ADS simultaneously takes advantage of trusted computing and virtualization technologies. Specifically, ADS allows a signature verifier to examine not only a signature's cryptographic validity but also its system security validity that the private signing key and the signing function are secure, despite the powerful attack that the signing application program and the general-purpose Operating System (OS) kernel are malicious. The modular design of ADS makes it application-transparent (i.e., no need to modify the application source code in order to deploy it) and almost hypervisor-independent (i.e., it can be implemented with any Type I hypervisor). To demonstrate the feasibility of ADS, we report the implementation and analysis of an Xen-based ADS system.

Index Terms—Data trustworthiness, digital signatures, cryptographic assurance, system-based assurance, malware

1 Introduction

Digital signatures are an important tool for ensuring data trustworthiness. The cryptographic assurance of digital signatures is well understood [1], assuming the private signing keys are not compromised (despite side-channel attacks [2]). An appreciated problem is to attain a stronger signature trustworthiness than the cryptographic assurance. However, existing solutions to this problem are not sufficient. Specifically, the cryptographic approach—including digital signatures of various flavors: threshold signature [3], proactive signatures [4], forward-secure signature [5], [6], key-insulated signature [7], and intrusion-resilient signatures [8]—can mitigate, but cannot prevent, the compromise of signature trustworthiness. PKI-like key revocation mechanisms are not sufficient because the compromise may not be detected until after a long time. It is also not sufficient to put the private signing keys in tamper-resistant hardware devices [9]. This is because the attacker can compromise the signing functions without compromising the private signing keys and without compromising the hardware devices; for example, the attacker uses stealthy malware to penetrate into the Operating System (OS) kernel and then asks the device to sign the attacker's messages [10], [11]. In order to enhance signature trustworthiness in the real world, we need to address such powerful attacks.

Our contributions. We propose enhancing data trustworthiness via *Assured Digital Signing* (ADS), which allows a signature verifier to examine not only digital signatures' cryptographic validity as in the current daily routine practice, but also their system security validity that the private signing keys and the signing functions are secure. In particular, ADS deals with the powerful attacks that the signing application program itself may be malicious (e.g., a backdoor was embedded by its vendor or developer), and that the underlying general-purpose OS kernel is malicious. In order to minimize the modifications to the Trusted Computing Base (TCB), we propose a modular design of ADS, which simultaneously takes advantage of trusted computing and virtualization technologies.

We show that ADS can enhance digital signatures' trustworthiness against the powerful attacks, as long as (essentially) the hypervisor is secure. Although this assumption is accepted by some researchers, it is deemed as somewhat strong by others. We believe that hypervisors will become significantly more secure in the near future (cf. [12], [13], [14], [15], [16]). The modular design of ADS makes it application-transparent because there is no need to modify the application source code in order to deploy it, and almost hypervisor-independent because it can be implemented on any Type I hypervisor. The modular security analysis, albeit informal, shows that if certain component properties are

- *W. Dai and H. Jin are with the Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: daiweiqi@gmail.com, hjin@hust.edu.cn.*
- *T.P. Parker is with the Department of Computer Science, College of Natural Sciences & Mathematics, Dallas Baptist University, 3000 Mountain Creek Pkwy., Dallas, TX 75211-9299. E-mail: paultparker@gmail.com.*
- *S. Xu is with the Department of Computer Science, University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249. E-mail: shxu@cs.utsa.edu.*

Manuscript received 7 Apr. 2011; revised 15 Sept. 2011; accepted 25 July 2012; published online 2 Aug. 2012.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSCSI-2011-04-0089.
Digital Object Identifier no. 10.1109/TDSC.2012.71.

satisfied by a hypervisor-specific instantiation of ADS, then the resulting instantiation attains the enhanced signature trustworthiness despite the powerful attacks. To demonstrate the feasibility of ADS, we report the implementation and analysis of an Xen-based ADS. We also discuss how to implement ADS on other popular Type I hypervisors.

Paper organization. The rest of the paper is organized as follows: Section 2 describes the design of ADS. Section 3 presents its instantiation on Xen hypervisor. Section 4 discusses how ADS may be instantiated on other Type I hypervisors. Section 5 reviews related prior work. Section 6 concludes the paper with future research directions.

2 Modular Design and Security of ADS

Now, we present the modular design of ADS. We argue that ADS attains the desired security properties based on that certain abstract component properties can be satisfied.

2.1 Design Requirements

The design requirements of ADS are: First, it should enhance the trustworthiness of digital signatures by allowing a signature verifier to examine not only the digital signatures' cryptographic validity but also their security validity that the private signing key and the signing function are secure. This naturally suggests using some form of attestation. Second, it should minimize the changes (if inevitable) to the TCB. Third, it should be applicable to most, if not all, system software platforms (i.e., platform independence) and should not force modification of the application source code in order to deploy it (i.e., application transparency).

The threat model consists of two attacks:

- Attack I: Both the signing application program and the general-purpose OS are malicious. The signing application program (e.g., electronic commerce or cloud computing) could be malicious by birth because its vendor or developer is malicious. We deal with general-purpose OS because it supports a rich set of (legacy) applications, but its security is much harder to guarantee, even in a foreseeable future.
- Attack II: The general-purpose OS is malicious but the signing application program is trusted (i.e., its vendor or developer is honest). In this case, the trusted signing application program may or may not be corrupted by the underlying general-purpose OS. For example, the malicious OS kernel may invoke the signing application program to obtain digital signatures.

Both attacks attempt to compromise the private signing keys, or to compromise the signing functions without compromising the private signing keys. As we will discuss in Section 5, existing solutions cannot defeat these attacks. We observe that Attack I is more powerful than Attack II. Nevertheless, we consider Attack II because it may be relevant in some real-life settings.

The security objective is therefore to ensure the following despite Attacks I-II:

- System Property 1: The signature verifier can verify that the private signing key is secure.
- System Property 2: The signature verifier can verify that the signing function is secure—the attacker cannot obtain unauthorized signatures.

The above means that we need some attestation service that is secure despite Attacks I-II.

Note that since the general-purpose OS kernel beneath the signing application is malicious, the attacker can always prevent the signing application from obtaining/producing signatures. In other words, this type of denial-of-service attack is inherent to the threat model. Since the denial-of-service attack causes little (if any) practical consequences on the trustworthiness of digital signatures, and the presence of this attack alerts that the system has been compromised and should be cleaned up, we do not consider this threat in the rest of the paper.

2.2 Design Rationales and Assumptions

In order to attain platform-independence while minimizing the (inevitable) modifications to the TCB, we propose adding a new software layer between the signing application program and the TCB. Since we cannot trust any general-purpose OS kernel, we propose using *Type I hypervisor* as (the major part of) the TCB, which resides on top of the bare hardware and below the general-purpose guest OS in the user Virtual Machines (VMs). The choice of using a Type I hypervisor rather than a Type II hypervisor, which runs on top of another software layer (e.g., a general-purpose OS such as Linux in the case of KVM [17]), is that the underlying software layer may be subject to extra attacks. Moreover, there are several popular Type I hypervisors [18], [19], [20], [21], [22].

In order to defeat the powerful attacks, we need to make the following security assumptions.

- Assumption I: The hypervisor is secure, as having been assumed in numerous studies. This is reasonable because there have been significant progress on making hypervisors more secure (e.g., [12], [13], [14], [15], [16]). Note that this implies that the BIOS is secure as well; otherwise, the hypervisor could be compromised.
- Assumption II: There is a secure light-weight kernel that can be used for some trusted VM (trusted-VM), whose hard disk storage integrity is also protected from any other VM. This is reasonable because the hypervisor is secure, and because the lightweight kernel can be substantially smaller than a general-purpose kernel and can even be formally verified to some extent [14].
- Assumption III: The relevant cryptographic primitives and their implementations (i.e., crypto libraries) are secure. These are well-accepted assumptions.

2.3 Modular System Design

The logical architecture of ADS is depicted in Fig. 1. In order to allow a signature verifier to examine the security validity of digital signatures, we propose extending the attestation service of Trusted Platform Module (TPM [23]) to accommodate some extra relevant information. Note that TPM's default attestation service only offers limited information (e.g., the system state before loading the hypervisor), which is necessary (otherwise, the hypervisor could be undermined)

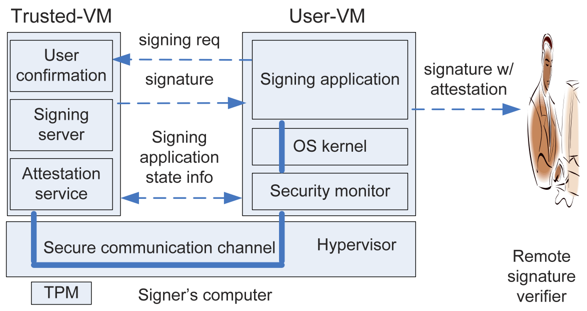


Fig. 1. Logical architecture of ADS, where dashed arrows represent logical (rather than physical) communication flows, and the thick solid line represents a secure communication channel between the application in the user-VM and the signing server in the trusted-VM.

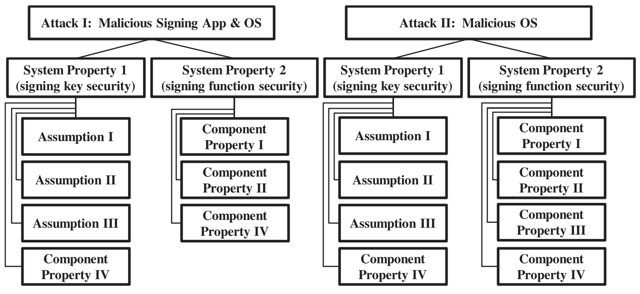


Fig. 2. Basic ideas behind how our modular design attains the desired system properties despite the attacks. Note that Component Property IV is essential to every System Property in the presence of any attack.

but not sufficient for our purpose. We need to additionally attest the signing application program, the message being signed and the resulting digital signature. We note that this is sufficient for our needs, but far from sufficient for attesting the integrity of the OS kernel or hypervisor, which is an open problem. The attestation of the signing application program is based on comparing its runtime hash measurement and its precomputed hash measurement, which may be provided by the software vendor or computed by the user in a clean state. The precomputed hash value is stored in trusted-VM's harddisk, and its integrity is protected from the malicious user-VM (as well as hard disk-oriented attacks, if applicable).

In order to deal with that the signing application itself may be malicious, we cannot let it generate or have access to the private signing key, and we must restrict its access to the cryptographic signing function. This led us to put the signing application in the untrusted user-VM (in which the OS kernel can be malicious), and to put a signing server in the trusted-VM. This separation is necessary, but still not sufficient, to defeat the attacks. We found it sufficient to give the human user a *user confirmation* mechanism to allow/deny a digital signing request. The user confirmation mechanism shows the relevant information about the message that is to be signed (e.g., file type, length, and even content for viewable document). Although this may require to run a document viewer for complicated documents, the addition to the trusted-VM TCB is small because the viewer does not need to support editing operations.

In order to allow the user-VM application program to use the signing service despite the malicious user-VM kernel, we propose adding a small piece of software, called *security monitor*, to reside below the user-VM kernel and above the hypervisor. This can reduce the reliance on the user confirmation mechanism to defeat the attacks, which is especially relevant when the attacker tampers the application program. This is important because we should weaken the reliance on the human user as much as possible (e.g., some users may not be careful enough). In order to attain secure communications between the application program and the signing server, we need a communication channel that can survive the malicious user-VM kernel. With the help of the security monitor, this is feasible because we only need to ensure communication *integrity*, namely that the messages are not manipulated.

Putting the above discussions together, we can see that at a high-level ADS operates as follows: First, both the signing server in the trusted-VM and the security monitor in the user-VM are initiated. Second, the signing application program requests the signing server to sign a message via the integrity-guaranteed communication channel, from which the signing application receives the resulting digital signature and an attestation. Third, the signature verifier can verify the cryptographic validity of the signature and the attestation, which would give the signature verifier extra confidence on the trustworthiness of the signature.

Discussion. First, although the hypervisor or the trusted-VM can (be enhanced to) do attestation, we choose TPM-based attestation because it is a mature technique with an embedded PKI-like infrastructure [24], and because it avoids unnecessary modifications to the TCB (i.e., hypervisor and trusted-VM) for attestation purpose. Second, we observe that TPM's attestation is essentially conducted through a digital signature with respect to TPM's own private signing key. Although one can exploit TPM for storing a user's private signing key (which is not the TPM's private key) and for signing applications, this does not solve the problem we address because TPM is used as a special hardware device, which can be invoked by the compromised OS kernel. This justifies the importance of the user confirmation mechanism. Third, we do not run the signing server in the hypervisor not only because this will significantly enlarge the hypervisor TCB, but also because hypervisor alone cannot provide the user confirmation mechanism (i.e., it has to be provided with some user interface). This explains why we provide the user confirmation mechanism via the trusted-VM.

2.4 System Properties of ADS-Produced Digital Signatures

Our objective is to ensure the signature verifiers that the private signing key is secure (System Property 1) and that the private signing function is secure (System Property 2). The basic ideas behind how the desired system properties are attained despite the attacks are highlighted in Fig. 2.

Specifically, in order to make our security analysis not specific to any hypervisor-specific instantiation (but rather easily applicable to various hypervisor-dependent instantiations), we relate the system properties to four abstract component properties, which are hypervisor-specific. As a corollary, any hypervisor-specific instantiation of ADS, which can be shown to satisfy the components properties, attains the desired system properties. The four abstract component properties are:

- Component Property I: The user confirmation mechanism allows the user to check the messages to be signed, and the user will carefully check the messages to be signed.

- Component Property II: If the signing application program in the user-VM is tampered by the underlying malicious kernel, this can be detected. This requires that the hash value of the application program be correctly computed despite the attacks.
- Component Property III: The malicious user-VM kernel cannot compromise (without being detected) the integrity of the communication channel between the signing server in the trusted-VM and the signing application in the user-VM, as long as the signing application is not malicious by birth (e.g., with an embedded backdoor).
- Component Property IV: The extended TPM-based attestation service is secure, where the extension is to attest the integrity of the signing application program, the message being signed, and the resulting signature.

In what follows, we argue that the above modular design attains System Properties 1 and 2 despite Attacks I-II, as long as the assumptions and the component properties are satisfied. We observe that Component Property IV reduces demonstrating the System Properties to showing that the private signing key and signing function are secure.

Assuring System Property 1 despite Attacks I-II. To see that the user's private signing key is not compromised by Attacks I-II, we observe that the key is generated in, and never leaves, the trusted-VM. Since both the trusted-VM (including its hard disk) and the underlying hypervisor are secure (Assumptions I and II) and the cryptographic primitives and their implementations are secure (Assumption III), the private signing key cannot be compromised by Attacks I-II.

Assuring System Property 2 despite Attack I. Now we argue that Attack I cannot compromise the signing function, namely that any access attempt from the attacker will be detected and prevented. Since we already showed that the attacker cannot compromise the private signing key, the attacker can only attempt to launch signing requests either from the malicious signing application, or from any other software program running in the user-VM. Because of Component Property I that the user will carefully check the messages to be signed, the attacker cannot trick the signing server to sign a message when requesting from the signing application program. Since the hash functions and the trusted-VM are secure, the attempt of requesting a signature from any other software program will be detected. This is because the hash value of the caller software program does not match the hash value of the authorized (albeit malicious by birth) signing application, where the former is guaranteed to be correctly measured by Component Property II and the latter is stored in the trusted-VM and cannot be compromised.

Assuring System Property 2 despite Attack II. Now we argue that Attack II cannot compromise the signing function. In this case, the user-VM kernel is malicious but the signing application is trusted. Since we already showed that the attacker cannot compromise the private signing key, the attacker can only attempt to obtain a signature

1. by invoking the authorized signing application without corrupting it,
2. by invoking any other software program running in the user-VM,
3. by corrupting the trusted signing application, and
4. by replacing the signing application's message with the attacker's.

As we argue below, these attempts are all prevented. Specifically, the above step 1 is prevented because the signing request will not be allowed by the user confirmation mechanism (Component Property I). The above step 2 is detected because the hash values do not match as mentioned in the case of Attack I. The above step 3 is detected and prevent because of Component Property II that a trusted signing application cannot be manipulated without being detected. The above step 4 is prevented because the malicious kernel cannot compromise the integrity of the communication channel (Component Property III). Note that the above steps 3 and 4 do not have counterparts under Attack I because the signing application is malicious by birth in Attack I, and explain why defeating Attack II will require less reliance on the user confirmation mechanism.

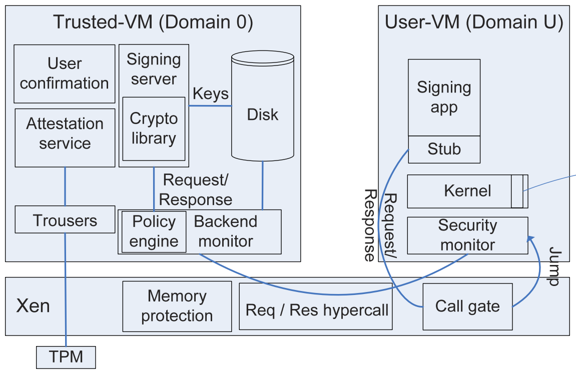


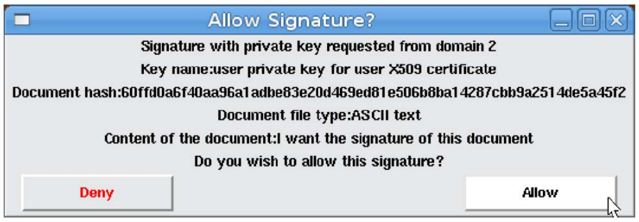
Fig. 3. Architecture of Xen-based ADS, where the main components—*user confirmation*, *signing server*, *attestation service*, and *security monitor*—are inherited from the logical architecture of ADS described in Fig. 1. The other components—*memory protection*, *request/response hypercalls*, *call gates*, *policy engine*, and *back-end monitor*—are specific to Xen hypervisor (but can be adapted to the other Type I hypervisors as we will discuss in Section 4).

3 Xen-Based Implementation of ADS

To demonstrate the feasibility of ADS, we conduct a case study based on paravirtualized Xen, in which Domain U never executes in Ring 0. Xen is chosen because it is widely used and its source code is freely available. This naturally leads to use Xen's Domain 0 and Domain U as the trusted-VM and the user-VM, respectively. As such, we will use "Domain 0" and "trusted-VM" interchangeably, and use "Domain U" and "user-VM" interchangeably. To be consistent with the assumptions made in the above logical design of ADS, we assume that Xen and Domain 0 are secure. This is a reasonable assumption because our focus here is to show the feasibility of ADS.

3.1 Architecture of Xen-Based ADS

Fig. 3 depicts the architecture of Xen-based ADS. It inherits the components of the logical ADS as shown in Fig. 1, while adding some Xen-based components. In order to make ADS application-transparent, we use a *stub* module to declare functions in the crypto library so that the application code can link against them just like linking against a local crypto library.



Signature with private key requested from domain 2
Key name:user private key for user X509 certificate
Document hash:60ffd0a6f40aa96a1adbe83e20d469ed81e506b8ba14287cbb9a2514de5a45f2
Document file type:ASCII text
Content of the document:I want the signature of this document
Do you wish to allow this signature?

Fig. 4. User confirmation mechanism in Domain 0: After the signing server in Domain 0 receives a signing request from Domain U, it pops up the dialog window in Domain 0 to ask the human user to allow/deny the signing request. In this concrete example the dialog window displayed information such as the message to be signed.

When the signing application in the user-VM needs to obtain a signature on a message, it uses a Xen-based communication channel, which instantiates the abstract secure communication channel in logical ADS architecture as depicted in Fig. 1, to communicate with the signing server in the trusted-VM. Since the user-VM kernel is malicious, the channel should not be based on the kernel and actually should be protected from the kernel. The channel will be fundamentally based on Xen's *shared memory* mechanism, with extra security protections that will be implemented by some special hypercalls we introduce. Since the signing application does not have the privilege to make hypercalls, it will need to use the *call gate* mechanism of x86 hardware to "jump" into the security monitor to make hypercalls (as illustrated in Fig. 3).

When the signing server receives the signing request, the policy engine will compare the hash value stored in Domain 0 for the authorized signing application and the hash value of the requesting program in Domain U, with the latter being measured by Xen in some careful fashion (essentially, all libraries the signing application calls are loaded into the memory and the memory pages are properly locked as we will see in Section 3.4). If they match, the signing server will pop up a dialog window in Domain 0 to ask the human user to allow/deny the signing request, which is the key idea to defeat the malicious signing application and to defeat the malicious Domain U kernel invoked signing request via the authorized signing application (cf. Fig. 4). Note that the user confirmation mechanism is in the trusted-VM and therefore protected from the malicious user-VM. If the human user allows the signing request, the signing server will send the signing application the resulting digital signature as well as an attestation. The attestation states that the signature was issued by a specific signing application, and that the hypervisor was loaded in a secure environment as the TPM-based bootstrapping chain indicated.

As discussed in the design of the logical ADS architecture, we wanted to minimize the modifications to the TCB and to use the TPM-based attestation infrastructure. Specifically, we use Trousers, which is the open-source implementation of TCG's Software Stack (TSS [23]), to allow the trusted-VM to access the TPM. In order to defeat the replay attack against attestation, the signature verifier can select a nonce and send it to the signing application, which will send the nonce to the attestation service so that the nonce will be signed by the attestation service. As a result, the signature verifier can verify, in addition to the cryptographic validity of the signature on the message in question, the hash value of the signing application program, which is stored in TPM's Platform Configuration Register (PCR) 9, and the hash value of the messages/signatures, which is stored in PCR 11. Note that PCR 9 and PCR 11 are used for our purpose because they were not used by the current TPM specification.

In what follow, we will elaborate the details related to: the establishment of the secure communication channel (Section 3.2), the security monitor and its protection (Section 3.3), the attestation service (Section 3.4), and the new hypercalls for securing ADS (Section 3.5). In particular, we mention that the new hypercalls allow Xen to measure the integrity of the signing application in the user-VM, while properly protecting the memory pages corresponding to the signing application program code segment, including both the executable and all of the libraries the application program calls (including shared libraries) without relying on the kernel or its data structures. Identifying these externally is feasible because in Linux x86's 32-bit Physical Address Extension (PAE) mode, all code-segment pages are marked as "NX == 0" if and only if they are `executable`. In summary, our modifications to Xen are the incorporation of a memory protection mechanism for preventing important memory regions from being tampered, and the introduction of the special hypercalls for facilitating secure communications between the signing application and the signing server. The total modifications to Xen is no more than 800 lines of C code. (The security monitor is 274 lines of C and assembly code.)

3.2 The Secure Communication Channel Component

We propose using Xen's *shared memory* mechanism to attain the secure communication channel between the signing application and the signing server. The reason for choosing this mechanism, rather than the *memory copy* mechanism, is not only that we can simplify the control over the protected content (i.e., one copy in the case of memory mapping versus multiple copies in the case of memory copy), but also that memory mapping is more efficient especially when the messages to be signed are in large volume.

As illustrated in Fig. 5, four physical memory regions, M_0-M_3, are used as shared memory. Since M_0-M_3 are mapped to the user-VM, we need to protect them by making the following changes to the page table entries (PTE) corresponding to M_0-M_3. Specifically,

- M_0: It contains the code of the security monitor and is always marked as `read-only` and `executable`.
- M_1: It stores parameters and hash-based measurements for attestation purpose (i.e., hash values of the signing application program, of the message to be signed, and of the resulting signature). M_1 is marked as `writeable` when a special hypercall needs to write to it, and `read-only` and `nonexecutable` otherwise.
- M_2: It is used by the signing application to write the message to be signed. It is always marked as `nonexecutable`, additionally marked as `writeable` before the signing application writes to it, and marked `read-only` afterwards.

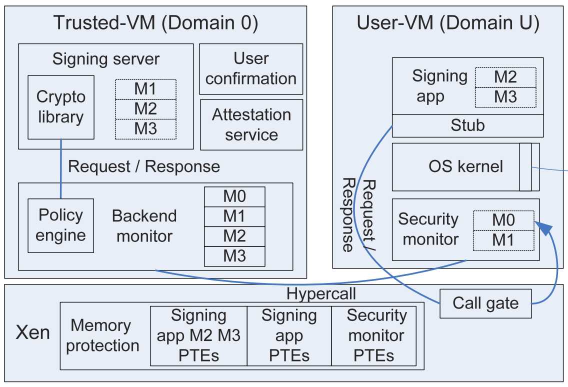


Fig. 5. Highlight of the "shared memory"-based implementation of secure communication channel. There are four physical memory regions, M_0-M_3, with each of size 256 KBytes (which can be set as a system parameter when installing ADS). M_0 and M_1 are mapped to the user-VM's kernel space for the security monitor itself, M_2 and M_3 are mapped to the user-VM's user space for the signing application, and M_1, M_2, and M_3 are mapped to the trusted-VM's user space for the signing server. M_0 stores the code of the security monitor, M_1 stores information for attestation, M_2 stores the message, and M_3 stores the signature.

- M_3: It is used by the signing server to return signature to the signing application. M_3 is always marked as `read-only` and `nonexecutable`.

Note that without special protection, the malicious user-VM kernel can use Xen's standard hypercalls, namely DO_MMU_UPDATE, DO_UPDATE_VA_MAPPING, and PTWR_DO_PAGE_FAULT, to manipulate the content of the shared memory regions. This is because each of the three hypercalls can be exploited to modify the page table entries in two fashions: 1) changing the `R/W` bit of the relevant page table entries to `writeable`; 2) creating a new page table entry and marking its `R/W` bit as `writeable` and mapping it to the protected memory region. As a consequence, the malicious kernel can arbitrarily tamper the content in the shared memory regions.

In order to defeat the attacks, a straightforward method is to use an extra list to record the protected memory pages. This is however costly because whenever the user-VM changes a page table entry, the system would have to search through the whole list to find out whether a certain memory page is protected. Instead, we use the following much more efficient method to prevent the kernel from tampering with the page table. Specifically, we use the 26th bit of `page->u.inuse.type_info` in Xen's `frame_table`, which was not used by Xen, to mark whether a page needs to be protected for the purpose of ADS. Our method is much faster because it totally avoids the operation of searching the list. We call the particular bit `PGT_entry_protected`. Similarly, we use the 25th bit of the corresponding `page->u.inuse.type_info` in Xen's `frame_table`, which was not used by Xen as well, to mark whether the page is map-protected. We call this bit `PGT_map_protected`.

3.3 The Security Monitor Component

The security monitor allows the signing application in the user-VM to use the new hypercall V, which will be introduced in Section 3.5, to establish a secure channel with the signing server in the trusted-VM. This channel is used for security sensitive operations that cannot be conducted via the kernel model, which may have been compromised by the kernel. Although using the call gate to jump to the security monitor to make hypercalls is not as efficient as using the kernel module to make hypercalls, we must use the security monitor to make hypercalls for the following security sensitive operations: mapping memory shared by the trusted-VM to its virtual address space; sending a message to the trusted-VM for signing; unmapping the shared memory when it is not needed any more. Specifically, the signing application invokes the new hypercall V through the following call gate mechanism:

- call_gate(`1,user_va`) where `user_va` is the virtual address of the signing application program: This call gate jumps to the security monitor to execute the new hypercall V to ask the hypervisor to measure the application program, set memory protection for the application program's executable pages, and map the shared memory region to `user_va` so as to pass the message and the signature.
- call_gate(`2,0`): This call gate jumps to the security monitor to execute the new hypercall V to ask the signing server in the trusted-VM to sign a message.
- call_gate(`3,0`): This call gate jumps to the security monitor to execute the new hypercall V to unmap the shared memory and remove the memory protection.

We stress that we cannot use the kernel module to do these security sensitive operations because of the following possible attack, which can be successfully launched by the malicious kernel unless the human user always carefully examines the messages to be signed. Suppose the signing application invokes call_gate(`1,user_va`) to jump to the kernel module, which invokes the new hypercall V as the security monitor does. Suppose the kernel module is compromised by the kernel,[1] it can change the second argument to a virtual address that points to the message of its choice, which will be mapped to M_2 and M_3. Since the attacker does not modify the application program, the hash-based integrity verification cannot detect this attack. As a consequence, the attacker can get a digital signature on the message of its choice, unless the user always carefully examines the message in the user confirmation window. The above attack is prevented by the security monitor because it is installed by the back-end monitor in the trusted-VM, and because its integrity cannot be corrupted by the user-VM kernel because of the aforementioned memory protection mechanism we added to Xen.

3.4 The Attestation Component

In order to measure the integrity of the signing application in the user-VM, we need to know what comprises the executable without relying on the user-VM kernel. Recall that we measure the hash of application the executable and the libraries (including the shared ones) the application program calls, which exploits that in x86's 32-bit PAE mode, all code-segment pages are marked as "$NX == 0$" if and only if they are `executable`. When the signing application program uses the call gate to request for signing service, the

1. There is no practical way, if feasible at all, for the hypervisor to ensure the integrity of the kernel module because the kernel module can make many system calls, which can be compromised by the kernel.

hypervisor will compute the hash value of the application program's executable pages, write the hash value in M_1, and notify the policy engine. The policy engine compares this measurement against the hash value of the application program stored in the trusted-VM, which may be provided by the software vendor or simply computed by the user at a clean system state. If they match, the policy engine extends this measurement into TPM's PCR 9 for later attestation purpose; otherwise, it prompts the user that the application program in the user-VM has been corrupted. After the user allows the signing request and the signing server generates a digital signature, the signing server measures the hash value of the message and the signature, and extends the hash value to TPM's PCR 11. Then, TPM's standard attestation is extended to accommodate PCR 9 and PCR 11, which allows the signature verifier to examine the signing application, the message and the signature. Note that all the hash values are computed using SHA-256.

For applications that require to sign multiple signatures, we propose the following cache-like performance enhancement. Specifically, we allow the signing application to request multiple signatures while amortizing the cost of measuring the signing application program's memory pages. This can be achieved by locking the application program's executable pages so as to prevent the Time-of-Computing-to-Time-of-Use (TOCTOU) attack that can tamper the binary after the measurement. As a result, the hypervisor only needs to compute the hash value of the application program once for signing multiple messages.

It is worthwhile to point out that if needed (e.g., when ADS is used as a component in larger system), the attestation service can be further extended to incorporate, for example, the system call table and the Interrupt Descriptor Table (IDT) in the user-VM. These tables do not change unless the kernel is re-compiled and are often modified by attacks, which means that this would allow detecting some attacks. If the signing application needs to modify the Page Table Entries, it can invoke call_gate(`3,0`) to unlock the memory. When the signing application needs to use the signing service the next time, it can invoke call_gate(`1,0`) to lock and measure the executable pages again.

3.5 New Hypercalls

Five new hypercalls are introduced. Before we describe the designs of the hypercalls in detail, we highlight how the hypercalls are used. The back-end monitor in the trusted-VM uses the new hypercall I to inform Xen about the shared memory so that the user-VM kernel module can map the shared memory to its address space via the new hypercall II. After establishing the shared memory, the back-end monitor uses the new hypercall III to add a call gate entry to Domain U's GDT (x86 Global Descriptor Table) so that the signing application can use the call gate to jump to the start address of the security monitor. The back-end monitor in the trusted-VM and the kernel module in the user-VM use the new hypercall IV to notify events to each other. (Note that we could instead use the security monitor, rather than the possibly compromised kernel module, to interact with the back-end monitor for event notification. However, this method is less efficient without offering significant security benefit.) The signing application in the user-VM uses the new hypercall V to interact with the trusted-VM signing server, which is part of the security communication channel.

New hypercall I. The back-end monitor initiates the shared memory by allocating four physical memory regions, whose start address is the hypercall argument. For security, this hypercall does the following to ensure that the shared memory can only be mapped by our newly introduced hypercalls and cannot be mapped using Xen's standard hypercall for sharing memory: We mark each shared memory page as map-protected. Moreover, we set the global flag `shared_memory`, which we introduced, to 1 to indicate the shared memory is ready to be mapped into the user-VM with the new hypercall II. Finally, we modify Xen's function GNTTAB_MAP_GRANT_REF to ensure that if `shared_memory != 0` and the page the user-VM kernel module (or any other program) requests to map is marked as map-protected, then Xen will disallow mapping to the shared memory page(s) no matter which hypercall made the request.

New hypercall II. The user-VM kernel module uses this hypercall to map/unmap the shared memory that was set up by the trusted-VM back-end monitor. The first argument is the map/unmap flag that indicates one of the following two possible scenarios: 1) The request is to map the shared memory and the second argument is `shared_pages_addr`, the start address of the shared memory. Xen sets `shared_memory` $:= 0$ before the mapping operation (so that mapping can be fulfilled), and sets `shared_memory := 2` after the mapping operation, which indicates that the shared memory cannot be remapped or unmapped partially via *any* hypercalls from the user-VM. 2) The request is to unmap the shared memory pages, and the second argument is not used. This is used when the application no longer needs the assured signing service. Xen allows the unmap operation only if `shared_memory == 2`, and disallows it otherwise.

New hypercall III. The trusted-VM back-end monitor uses this hypercall to add an entry for storing the start address of M_0 to Domain U' GDT, which can only be changed by Xen. The start address of M_0 is fixed after the user-VM maps the shared memory, and is stored in Xen's global variable `shared_pages_addr`. After adding the entry to GDT, the back-end monitor copies the security monitor code, which is stored in the trusted-VM, to the shared memory M_0-M_1. As a result, the call gate can jump to the start address of M_0, namely the start address of the security monitor.

New hypercall IV. This hypercall allows the trusted-VM and the user-VM to use the following six newly introduced virtual interrupts to notify each other of events. This hypercall has one argument that indicates one of the following operations:

1. The user-VM kernel module uses this to notify the trusted-VM that the signing application is starting up.
2. The trusted-VM back-end monitor uses this to notify the user-VM that it has set up the shared memory.
3. The user-VM kernel module uses this to notify the trusted-VM that it has mapped the shared memory.
4. The trusted-VM back-end monitor uses this to notify the user-VM that it has set up the call gate.
5. The trusted-VM back-end monitor uses this to notify the user-VM that the signature for the message

provided by the signing application is ready for picking up, and M_2 is labeled as `writeable`.

6. The user-VM kernel module uses this to notify the trusted-VM that the signing application is being teared down.

New hypercall V. This hypercall has two arguments, and allows the application to perform privileged operations specified by the first argument, which has three possibilities. First, when the application wants to map shared memory M_2-M_3 to its virtual address `user_va`, it invokes call_gate(`1, user_va`), which causes the hypercall to do the following:

1. Use SIGPROCMASK() to disable software interrupts so as to prevent interference and ensure that the kernel cannot gain control over the CPU.
2. Lock the application's executable pages by marking them as `read-only` (if not already) and mark each page memory-protected(indicating memory protection). As a result, the application executable cannot be modified.
3. Set M_1 as `writeable`, hash the application executable pages and record the measurement in M_1, and set M_1 as `read-only` again.
4. Modify the page tables so that the application can map M_2-M_3 to its virtual address that is given by the input argument. (Note this hypercall can operate directly on the page table of the application because CR3's value and the application's page table do not change during the invocation process.) Set M_3 as `read-only` and M_2 as `writeable`, and enable software interrupt.

Second, when the signing application wants to request the signing service for signing a message, it invokes call_gate(`2, 0`), which causes the hypercall to do the following: 1) Use SIGPROCMASK() to disable software interrupts so as to prevent interference. Set M_2 as `read-only` so that the message cannot be tampered with. 2) Set M_1 as `writeable`. Write to M_1 the argument `(2,0)`. Set M_1 as `read-only` again, and enable software interrupt.

Third, when the application wants to unmap M_2-M_3 (i.e., no longer needs the service), it invokes call_gate(`3, 0`), which causes the hypercall to do the following: 1) Use SIGPROCMASK() to disable software interrupts so as to prevent interference. Unmap M_2-M_3 from the application. 2) Remove the memory protection on the application executable pages. Finally, enable software interrupts.

3.6 Running Example

Having described the individual components in some degree of details, now we show how the components are put together through a running example. Fig. 6 demonstrates a running example of message flows, with details elaborated below.

0. The Domain 0 back-end monitor (a kernel module) and Domain U kernel module are loaded into the trusted-VM and the user-VM, respectively.
1. Both kernel module devices are opened and registered to be ready for handling virtual interrupts.
2. The Domain U kernel module raises a virtual interrupt to notify the back-end monitor that the signing application is set up.
3. Upon receiving the interrupt, the back-end monitor allocates the shared memory pages.
4. The back-end monitor uses the new hypercall I to inform Xen that the shared memory pages are to be shared with Domain U, and uses a virtual interrupt to notify the Domain U kernel module.
5. Upon receiving the interrupt, the Domain U kernel module invokes the new hypercall II to map the shared memory, and uses a virtual interrupt to notify the back-end monitor that the mapping has been accomplished.
6. Upon receiving the interrupt, the back-end monitor uses the new hypercall III to modify GDT and install the security monitor, and uses a virtual interrupt to notify the Domain U kernel module that the call gate is ready for use. At this point, the system initialized.
7. The application calls call_gate(`1,user_va`) to invoke the new hypercall V, which maps M_2 and M_3 to the memory region that starts at `user_va`.
8. The back-end monitor maps M_1-M_3 to the address space of the signing server. The policy engine verifies the measurement of the application executable against the one stored in Domain 0, and locks the executable pages so they cannot be changed.
9. The application copies the message into M_2.
10. The application invokes call_gate(`2,0`) to execute the new hypercall V to notify crypto service that a message is waiting to be signed.
11. The signing server reads the message from M_2. The policy engine pops up a dialog window for the user to allow the signing. When the signature is ready, the signing server computes the hash value of the concatenation of the message and the signature. The back-end monitor places the signature into M_3, and then uses virtual interrupt to notify the kernel module in Domain U that the signature is ready for picking up.
12. Upon receiving the interrupt, M_2 becomes `writeable` and the signing application reads the signature from M_3.
13. If the signing application wants to send another message for signing, it returns to Step 9.
14. When the signing application no longer needs the signing service, it calls call_gate(`3,0`) to invoke the new hypercall V, which notifies the back-end monitor that the service is no longer needed. The application closes the device file that is connected to the kernel module.
15. In order to tear down the assured signing service, the Domain U kernel module uses the new hypercall II to unmap M_0-M_3, and uses a virtual interrupt to notify the back-end monitor. Upon receiving the interrupt, the back-end monitor unmaps the M_1-M_3 that were mapped to the crypto service, closes the device file, and destroys the shared memory.

3.7 How Are the Component Properties Attained by the Xen-Specific Instantiation?

In Section 2.4, we argued that any hypervisor-specific instantiation of ADS attains the desired System Properties 1 and 2 as long as the three assumptions and the four component properties are satisfied. This means that in order to show the Xen-specific instantiation of ADS attains System Properties 1 and 2, we only need to show that the Xen-based ADS instantiation has Component Properties I-IV. Fig. 7

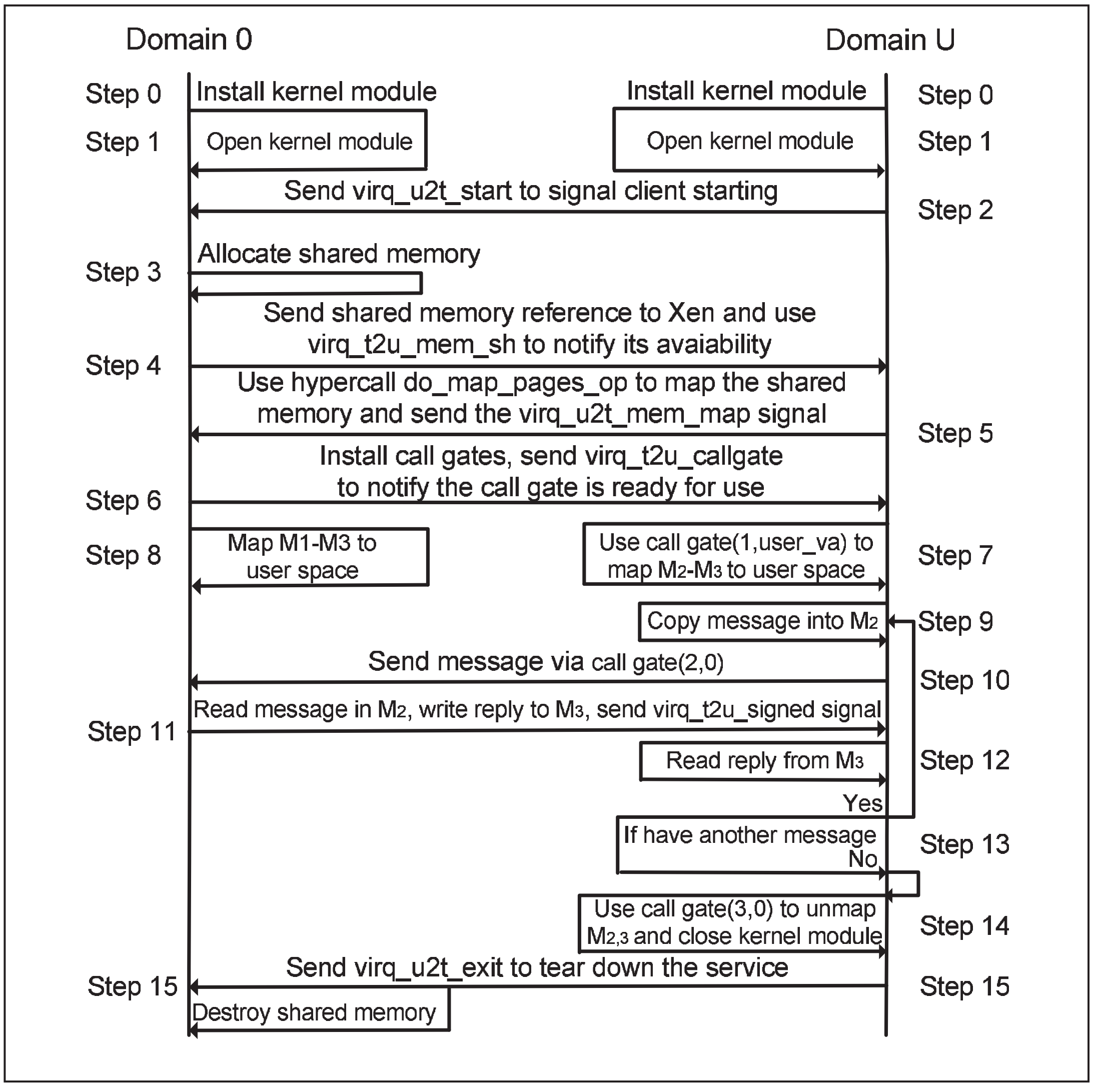

Domain 0
Domain U
Step 0
Install kernel module
Install kernel module
Step 0
Step 1
Open kernel module
Open kernel module
Step 1
Send virq_u2t_start to signal client starting
Step 2
Allocate shared memory
Step 3
Send shared memory reference to Xen and use virq_t2u_mem_sh to notify its avaiability
Step 4
Use hypercall do_map_pages_op to map the shared memory and send the virq_u2t_mem_map signal
Step 5
Install call gates, send virq_t2u_callgate to notify the call gate is ready for use
Step 6
Step 8
Map M1-M3 to user space
Use call gate(1,user_va) to map M2-M3 to user space
Step 7
Copy message into M2
Step 9
Send message via call gate(2,0)
Step 10
Read message in M2, write reply to M3, send virq_t2u_signed signal
Step 11
Read reply from M3
Step 12
Yes
If have another message
No
Step 13
Use call gate(3,0) to unmap M2,3 and close kernel module
Step 14
Send virq_u2t_exit to tear down the service
Step 15
Step 15
Destroy shared memory

Fig. 6. A high-level description of the system control flow in the Xen-based ADS system, where Steps 0-6 correspond to *system initialization*, Steps 7-8 correspond to *application initialization*, Steps 9-13 correspond to *digital signing*, and Steps 14-15 correspond to *system teardown*.

highlights the basic ideas behind how they are attained despite the attacks. The detailed discussions are described below.

Assuring Component Property I despite Attacks I-II. This property is that the user confirmation mechanism allows the user to examine the message that is to be signed. As a result, any signing request with a message that is not intended by the user will be detected via the user confirmation mechanism. Since the confirmation window belongs to Domain 0, which is assumed to be secure and further protected by the hypervisor from Domain U, the Domain U kernel cannot simulate or produce a software-based click in Domain 0. The dialog window shows information such as the message itself (if it is viewable). A caveat in the above reasoning is that a careless user may always click the Allow button in the user confirmation window (cf. Fig. 4). This means that taking full advantage of our solution would need to raise average users' awareness, which is common to most technical security solutions. As long as a user has a reasonable degree of awareness, our solution is resilient.

Assuring Component Property II despite Attacks I-II. This property is that if the signing application is corrupted, this attack can be immediately detected. Suppose the malicious kernel tampered the signing application. Then, the policy engine will detect this attack when comparing the hash value of the authorized signing application program, which was stored in Domain 0, and the runtime hash value of the caller program, which is measured by the hypervisor after properly locking the corresponding memory pages. Since Domain 0 (including its hard disk for storing the hash

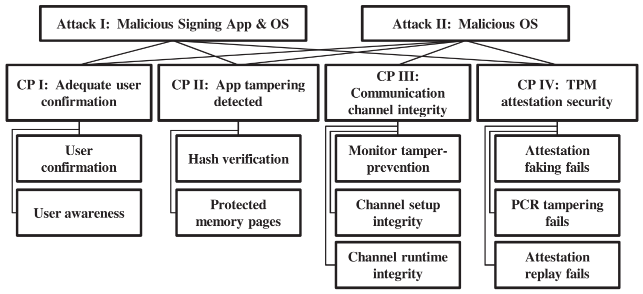

Attack I: Malicious Signing App & OS
Attack II: Malicious OS
CP I: Adequate user confirmation
CP II: App tampering detected
CP III: Communication channel integrity
CP IV: TPM attestation security
User confirmation
User awareness
Hash verification
Protected memory pages
Monitor tamper-prevention
Channel setup integrity
Channel runtime integrity
Attestation faking fails
PCR tampering fails
Attestation replay fails

Fig. 7. Basic ideas behind how our carefully designed Xen-specific mechanisms help attain the desired component properties despite the attacks, where CP stands for Component Property.

values of authorized signing applications) is secure, the policy engine will detected the attack. Moreover, after the measurement is conducted, the memory pages of the signing application are always protected until it is not needed. As a corollary, if the malicious kernel uses any other software program to request the signing service, this kind of impersonation will be immediately detected.

Assuring Component Property III despite Attack II. (Attack I is not applicable in this case.) This property is that the integrity of the communication channel between the (trusted) signing application and the signing server is guaranteed. Since Component Property II already showed that any modification to the signing application will be immediately detected by the policy engine, we only need to show the following:

1. The security monitor cannot be tampered.
2. The malicious kernel cannot corrupt the setup process of the communication channel.
3. The malicious kernel cannot corrupt the established communication channel.

To see the above part 1, we show the security monitor cannot be tampered as follows.

- The security monitor's memory pages cannot be tampered with by the Domain U kernel or any application running on top of it. This is because these pages are always marked as `read-only` to Domain U, and can only be modified by the new hypercall V that cannot be invoked by the kernel.
- The kernel cannot gain control over the CPU when the security monitor executes (e.g., by scheduling a timer interrupt) because the interrupts have been masked by Xen during the execution of the security monitor. While some system management interrupts (e.g., power events) are not maskable and hence cannot be disabled, exploiting such attacks require modifying the BIOS code, which is assumed to be secure (otherwise, the hypervisor could be compromised).
- During the execution of security monitor, the attacker cannot regain control by causing VM faults because the security monitor's memory is owned by Domain 0.

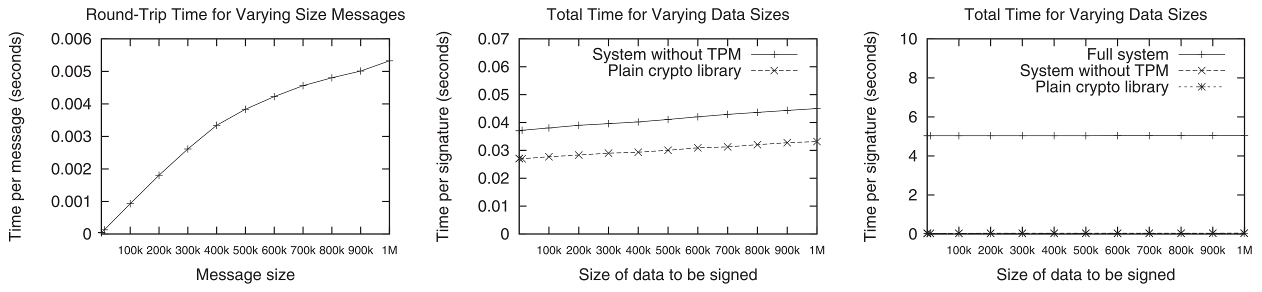
To see the above part 2, we show that the setup process of the communication channel is secure.

- The malicious Domain U kernel cannot map the shared memory with any method other than the new hypercall II. This is because our modification to Xen ensures that only the new hypercall II can map the shared memory. In particular, the modification to Xen prevents Domain U from using the standard `do_grant_table_op` hypercall to map the shared memory pages.
- The malicious Domain U kernel cannot map portion of the shared memory (i.e., the mapping must be with respect to the shared memory regions as a whole). This is achieved by Xen setting the `PGT_map_protected` bit to mark the pages as `map-protected`.
- The malicious Domain U kernel can neither unmap nor remap (any portion of) the shared memory after we map it. This is because after using our hypercall, the `shared_memory` flag is changed to two (mapped), which prevents Domain U from remapping the shared memory—even using our hypercalls—to another virtual address. Moreover, using the `do_grant_table_op` hypercall cannot map or unmap part of the shared memory somewhere else until after Xen tears down the shared memory (which occurs after the new hypercall II unmaps the shared memory), because `do_grant_table_op` checks for `shared_memory`=0.

To see the above part 3, we show that after the communication channel setup, all parts of the channel are secure during the runtime.

- The attacker cannot attack the hypercalls. The hypercall code is in Xen space, so Domain U cannot attack it. Moreover, the kernel may not intercept or fake the new hypercall II to map the shared memory to its address space; otherwise, the attempt will be detected by call_gate(`1,0`), which will report failure after noticing that the shared memory was never mapped because `shared_memory != 2`.
- The attacker cannot attack the call gate. The call gate entry can only be changed by Xen. Moreover, only Domain 0 can use the new hypercall III to add or change the entry for storing the start address of M_0 to GDT.
- The attacker cannot attack the memory protection module because it is in Xen space. Since the setup process of the communication channel is already shown to be secure, the attacker cannot cheat the memory protection module into protecting memory pages other than the shared memory regions.
- The attacker cannot attack the shared memory. This is because the modifications to Xen ensure that the Domain U kernel neither can modify the PTE's of Domain U's M_0-M_3, nor can modify the application's executable pages to let the PTE's map to the kernel's pages or to make them `writeable`. Since Xen marked the `PGT_entry_protected` bits for these pages before Domain U maps the shared memory, the kernel cannot tamper with these page table entries.
- The Domain U kernel cannot attack the page table entries pointing to the application's executable pages because the pages are protected by the hypervisor via the `PGT_entry_protected` bit.

Assuring Component Property IV despite Attacks I-II. This property says that if a TPM attests that a signature was requested by a particular application in the assured signing system, then it must have been the case. To compromise this assurance, there are three possible attacks, which will fail as we now explain. First, the attacker attempts to fake an attestation. This is infeasible because the attestation key is (created by and) kept within the boundary of the TPM, and the cryptographic scheme for attestation is assumed to be secure. Second, the attacker attempts to tamper TPM's PCR values. This is infeasible because the relevant PCR-related commands are sent by trousers in Domain 0. Moreover, measurement of the signing application is generated by Xen hypervisor, which also marks the memory storing the measurement as `read-only` to Domain U. Third, the attacker reuses or replays some past attestation. This is defeated because the signature verifier chooses a fresh random nonce for each attestation. As long as the nonce is sufficiently long, the probability that replay attack succeeds is negligible.



(a) ADS inter-VM communication time (b) ADS response time without attestation (c) ADS response time with attestation

Fig. 8. The inter-VM communication time versus ADS response time without attestation versus ADS response time with attestation.

3.8 Performance Evaluation

The performance of the Xen-based ADS is mainly influenced by three major factors: the time spent on the inter-VM communication, the time spent on the digital signing, and the time spent on attestation. In order to see which part is the most time consuming, We first measure the time spent on the inter-VM communication, then the ADS response time but without attestation, finally the ADS response time with attestation.

All experiments are performed on an HP xw4550 workstation, with a quad-core 2.3 GHz AMD Opteron processor and 4 GBytes of RAM. The machine has a v1.2 Broadcom TPM, revision level A2. The software environment for all experiments is paravirtualized Xen 3.3.1 installed on Ubuntu 8.04 LTS with the 2.6.18.8-xen kernel. The user-VM runs the same paravirtualized kernel that is provided with Xen 3.3 and we give the Domain U 512 M RAM and one virtual CPU core. For the signing server, we use Peter Gutmann's `cryptlib` library version 3.3.3. We use PKCS#7 CMS signing with RSA + SHA hash. The RSA key size was the cryptlib default (2,048 bits).

Fig. 8a plots the inter-VM communication time with varying size of messages, including the cost for protecting M_2. Each data point is averaged based on the 100,000 messages of the same size. We observe that for messages with size up to 500 KBytes, the time is linearly related to the message size; for messages with size exceeding 500 KBytes, the time is still linearly related to the message size but with a smaller factor. This phenomenon is likely that the AMD Opteron processor has 4×512 KBytes L2 Cache, and the CPU optimization will give priority to large (greater than 512 KBytes) and consecutive memory access, which would occur in our experiment.

Fig. 8b plots the response time of ADS but without attestation, where each point in the graph is averaged over 500 runs. For comparison purpose, it also plots the signing cost with local cryptography library. We observe that the differential is steadily about 0.01 seconds, which accommodates the 0-0.006 seconds depicted in Fig. 8a, the time spent on computing the hash value of the signing application, the time spent on mapping the shared memory regions, and the time spent on unmapping the shared memory regions after obtaining the signature.

Fig. 8c plots ADS's response time with TPM-based attestation, where each point in the graph is averaged over 500 runs. For comparison purpose, it also plots ADS' response time without attestation. We observe that there is a large performance difference (about 5 seconds) between using attestation and not using attestation. This is caused by the underlying attestation technique and specific to the TPM hardware platform in our experimental system. Specifically, the noticeable cost is due to the TPM quote operation for attestation, which is essentially for computing a 2,048-bit RSA signature on the relevant PCR values, which takes about 5 seconds on the TPM's low-end/cheap processor.

In summary, the inter-VM communication mechanism is indeed very efficient, which justifies our design choice of using the shared memory mechanism for this purpose. The phenomenon that ADS' response time is reasonably flat with respect to varying size of messages is caused by 1) the inter-VM communication mechanism is very efficient and only consumes a very small portion of the response time, and 2) the TPM-specific attestation service consumes most part of ADS' response time but is *independent* of the size of the messages because the attestation is always on the fixed-size PCR values. We believe that the computational cost of TPM-based attestation is not as significant as the security gain implied by not causing many modifications to the hypervisor (which would offer better performance). Moreover, any performance improvement in TPM's attestation service will directly benefit ADS.

4 Implementing ADS on Other Type I Hypervisors

As mentioned in Section 2, there are several other popular Type I hypervisors, including Citrix XenServer [19], VMware ESX/ESXi [20], [21] and Microsoft Hyper-V [22]. Since Citrix XenServer is based on the open source Xen, on which we implemented ADS, it would be straightforward to adapt our Xen-specific ADS to this platform. In what follows, we briefly discuss how ADS may be instantiated on the other two hypervisor platforms. This information is particularly useful to the developers who have access to the source code of these hypervisors.

In order to instantiate ADS on VMware's VMkernel hypervisor platform, the core technique for establishing secure communication channel between the user-VM and the trusted-VM would still be the shared memory mechanism. Moreover, one needs to implement the counterparts of the five new hypercalls we implemented for Xen. Note that Hypercalls in Xen are called Virtual Machine Interface (VMI) in VMware. In order to protect the shared memory from being attacked by the malicious user-VM kernel via VMware's VMI_ALLOCATEPAGE, VMI_RELEASEPAGE, VMI_SETPXE, and VMI_SWAPPXE, the reserved bit of *page info* in the VMkernel can be used to mark whether a page needs to be `write-protected` for the purpose of ADS.

Moreover, the other reserved bit of the corresponding *page info* in the VMkernel can be used to mark whether a page is `map-protected`.

In order to instantiate ADS on Microsoft Hyper-V hypervisor platform, the core technique for establishing secure communication channel between the user-VM and the trusted-VM would still be the shared memory mechanism. Since it already provides hypercall HVASSERTVIRTUALINTERRUPT for inter-VM events notification, one only needs to implement the counterparts of the other four new hypercalls. Similarly, in order to deal with that the malicious user-VM kernel can use the standard HVDEPOSITMEMORY, HVWITHDRAWMEMORY, HVMAPGPAPAGES, and HVUNMAPGPAPAGES hypercalls to attack the shared memory, one can use the reserved bit of the *memory access messages* in Hyper-V to mark whether a page needs to be `write-protected` and use the other reserved bit to mark whether the page is `map-protected`.

5 Related Work

The problem of enhancing the trustworthiness of digital signatures has led to many solutions, including a set of novel cryptographic notions [2], [3], [4], [5], [6], [7], [8], [11]. These solutions can mitigate the damage caused by the compromise of the private signing keys, but have very limited success against the compromise of signing functions (i.e., the keys are not compromised). ADS is based on a novel integration of trusted computing and virtualization technologies and can prevent the compromise of both the private signing keys and the signing functions.

ADS is related to *software-based* solutions to running sensitive applications in secure execution environment (e.g., [25], [26], [27], [28]) and to kernel-based application protections [29], [30]. ADS also shares some common characteristics with the recent theme of protecting sensitive applications from malicious OS [31], [32] because ADS's threat model accommodates the malicious user-VM kernel. However, all these solutions cannot deal with malicious applications and cannot deal with unauthorized execution of the security sensitive applications. In contrast, ADS addresses such threats by putting the human user in the loop (i.e., the user confirmation mechanism in the trusted-VM).

ADS is further related to *hardware-based* solutions to running sensitive applications in secure execution environment, such as the Flicker system [33], which extends Intel's Trusted Execution Technology (TXT) [34] and AMD's Secure Virtual Machine (SVM) [35] to provide a hardware-protected clean execution environment without rebooting the system. When Flicker is used for digital signing purpose, it is fundamentally similar to a tamper-resistant hardware device and therefore has the weakness that it can be invoked by a malicious OS kernel to sign any data, namely that the signing function is compromised although the private signing key is secure [10]. Moreover, it is not clear how Flicker can be extended to authenticate that the invocation is from a human user rather than a malicious kernel. Furthermore, Flicker assumes that the sensitive application is trusted; whereas, ADS explicitly deals with malicious signing applications. To be fair, ADS has its own limitation, especially the assumption that the hypervisor is secure. Fortunately, there have been many studies on enhancing hypervisor security (e.g., [12], [13], [14], [15], [16]). Recent more radical approaches include: letting VM's run on (almost) naked hardware so as to eliminate the hypervisor attack surface [36], exploiting specific hardware-enforced isolation for running workloads [37]; exploiting hardware-enforced access control to protect VM's from a malicious hypervisor [38].

6 Conclusion

Enhancing signature trustworthiness is of fundamental importance. We have presented a novel solution to this problem, while dealing with malicious signing application and/or malicious general-purpose OS kernel. To demonstrate its feasibility, we reported a Xen-based implementation and discussed the implementation issues on other Type I hypervisor platforms.

Future research directions. There are opportunities for future research. First, it is important to reduce, if not eliminate, the reliance on the user to confirm that a message signing operation was indeed initiated by a human user. Second, we observe that the user confirmation mechanism offers enhanced source nonrepudiation, which could be exploited to hold insiders (i.e., corrupt authorized users) more accountable. This is because they cannot attribute signatures to the attacks we addressed. As such, it is interesting to characterize to what extent accountability can be gained by deploying ADS. Third, it would be very interesting to formalize the type of (modular) security analysis we conducted. It appears that both the cryptographic framework and the Dolev-Yao framework are not applicable to this type of analysis. One challenge is that the security properties very much depend on the implementation details. Fourth, it would be interesting to extend ADS to accommodate more cryptographic applications.

Acknowledgments

The authors sincerely thank the anonymous reviewers for their comments that made the paper significantly better. They thank Peter Gutmann for his cryptlib library and for his support when they were trying to understand some of its more esoteric features. W. Dai was partly supported by National Base Research Program of China (2007CB310900) and National Science Foundation of China (60973038), and by an AFOSR MURI grant to University of Texas at San Antonio (UTSA) when he was visiting UTSA. P. Parker and S. Xu were partly supported by an AFOSR grant and an AFOSR MURI grant to UTSA. H. Jin was supported by National Base Research Program of China (2007CB310900) and National Science Foundation of China (60973038).

References

[1] S. Goldwasser, S. Micali, and R. Rivest, "A Digital Signature Scheme Secure against Adaptive Chosen-Message Attacks," *SIAM J. Computing,* vol. 17, pp. 281-308, Apr. 1988.

[2] A. Akavia, S. Goldwasser, and V. Vaikuntanathan, "Simultaneous Hardcore Bits and Cryptography against Memory Attacks," *Proc. Sixth Theory of Cryptography Conf. Theory of Cryptography (TCC),* O. Reingold, ed., pp. 474-495, 2009.

[3] Y. Desmedt and Y. Frankel, "Threshold Cryptosystems," *Proc. Ninth Ann. Int'l Cryptology Conf. Advances in Cryptology,* pp. 307-315, 1990.

[4] R. Ostrovsky and M. Yung, "How to Withstand Mobile Virus Attacks (Extended Abstract)," *Proc. 10th Ann. ACM Symp. Principles of Distributed Computing (PODC '91)*, pp. 51-59, 1991.

[5] R. Anderson, "On the Forward Security of Digital Signatures," technical report, 1997.

[6] M. Bellare and S. Miner, "A Forward-Secure Digital Signature Scheme," *Crypto '99: Proc. 19th Ann. Int'l Cryptology Conf. Advances in Cryptology,* M. Wiener, ed., pp. 431-448, 1999.

[7] Y. Dodis, J. Katz, S. Xu, and M. Yung, "Strong Key-Insulated Signature Schemes," *Proc. Sixth Int'l Workshop Theory and Practice in Public Key Cryptography (PKC '03)*, pp. 130-144, 2003.

[8] G. Itkis and L. Reyzin, "SiBIR: Signer-Base Intrusion-Resilient Signatures," *CRYPTO '02: Proc. 22nd Ann. Int'l Cryptology Conf. Advances in Cryptology*, pp. 499-514, 2002.

[9] B. Yee, "Using Secure Coprocessors," PhD thesis, Carnegie Mellon Univ., May 1994.

[10] 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," *Proc. 21st Nat'l Information Systems Security Conf. (NISSC '98)*, 1998.

[11] S. Xu and M. Yung, "Expecting the Unexpected: Towards Robust Credential Infrastructure," *Proc. Int'l Conf. Financial Cryptography and Data Security (FC '09)*, Feb. 2009.

[12] Z. Wang and X. Jiang, "Hypersafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," *Proc. IEEE Symp. Security and Privacy*, pp. 380-395, 2010.

[13] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky, "Hypersentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity," *Proc. ACM Conf. Computer and Comm. Security,* pp. 38-49, 2010.

[14] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D.C.P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal Verification of an os Kernel," *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles (SOSP '09)*, 2009.

[15] U. Steinberg and B. Kauer, "Nova: A Microhypervisor-Based Secure Virtualization Architecture," *Proc. Fifth European Conf. Computer systems (EuroSys '10)*, 2010.

[16] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," *Proc. IEEE Symp. Security and Privacy*, 2010.

[17] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: The Linux Virtual Machine Monitor," *Proc. Linux Symp.*, 2007.

[18] C. Systems, "Xen Project." http://www.xen.org/, 2011.

[19] C. Systems, "Citrix Xenserver." http://www.citrix.com/xenserver, 2011.

[20] R. Oglesby and S. Herold, *VMware ESX Server: Advanced Technical Design Guide,* Advanced Technical Design Guide Series. The Brian Madden Company, 2005.

[21] C. Chauba, "The Architecture of Vmware Esxi," VMware White Paper, 2008.

[22] Microsoft, "Windows Server 2008 r2 Hyper-V." http://www.microsoft.com/en-us/server-cloud/windows-server/hyper-v.aspx, 2010.

[23] T.C. Group https://www.trustedcomputinggroup.org/, 2012.

[24] E. Brickell, J. Camenisch, and L. Chen, "Direct Anonymous Attestation," *Proc. 11th ACM Conf. Computer and Comm. Security (CCS '04)*, pp. 132-145, 2004.

[25] R. Ta-Min, L. Litty, and D. Lie, "Splitting Interfaces: Making Trust between Applications and Operating Systems Configurable," *Proc. Seventh Symp. Operating Systems Design and Implementation (OSDI '06)*, pp. 279-292, 2006.

[26] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," *ACM SIGOPS Operating Systems Rev.*, vol. 37, no. 5, pp. 193-206, Oct., 2003.

[27] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," technical report, EECS Dept., Univ. of California, Berkeley, Feb. 2009.

[28] H. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: Rapid Virtual Machine Cloning for Cloud Computing," *Proc. Fourth ACM European Conf. Computer Systems (EuroSys '09)*, pp. 1-12, 2009.

[29] K. Xu, H. Xiong, C. Wu, D. Stefan, and D. Yaoxyd, "Data-Provenance Verification for Secure Hosts," *IEEE Trans. Dependable and Secure Computing,* vol. 9, no. 2, pp. 173-183, Mar./Apr. 2012.

[30] H. Almohri, D. Yao, and D. Kafura, "Identifying Native Applications with High Assurance," *Proc. ACM Conf. Data and Application Security and Privacy (CODASPY '12)*, 2012.

[31] X. Chen, T. Garfinkel, E.C. Lewis, P. Subrahmanyam, C.A. Waldspurger, D. Boneh, J. Dwoskin, and D.R. Ports, "Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems," *ASPLOS XIII: Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 2-13, 2008.

[32] J. Yang and K. Shin, "Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis," *Proc. Fourth Int'l Conf. Virtual Execution Environments (VEE '08)*, pp. 71-80, 2008.

[33] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for Tcb Minimization," *Proc. ACM European Conf. Computer Systems (EuroSys '08)*, 2008.

[34] Intel, "Intel Trusted Execution Technology mle Developers Guide," http://www.intel.com/technology/security/, June 2008.

[35] AMD, "Amd64 Virtualization: Secure Virtual Machine Architecture Reference Manual,"AMD Publication no. 33047 Rev. 3.01, May 2005.

[36] J. Szefer, E. Keller, R. Lee, and J. Rexford, "Eliminating the Hypervisor Attack Surface for a More Secure Cloud," *Proc. ACM Conf. Computer and Comm. Security,* pp. 401-412, 2011.

[37] A. Azab, P. Ning, and X. Zhang, "Sice: A Hardware-Level Strongly Isolated Computing Environment for x86 Multi-Core Platforms," *Proc. 18th ACM Conf. Computer and Comm. Security (CCS '11)*, pp. 375-388, 2011.

[38] J. Szefer and R. Lee, "Architectural Support for Hypervisor-Secure Virtualization," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

Weiqi Dai is working toward the PhD degree of computer science and technology at Huazhong University of Science and Technology (HUST) in China. His research is focused on virtualization technology and trusted computing.

T. Paul Parker received the undergraduate degree from Baylor University in computer science, the master's degree from Rice University in computer science, and the PhD degree in computer science from the University of Texas at San Antonio. He is an assistant professor at Dallas Baptist University, Texas. His primary research interests include the intersections of systems and security.

Hai Jin received the PhD degree in computer engineering from HUST in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He is now a dean of the School of Computer Science and Technology at HUST. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He was at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. His research interests include computer architecture, virtualization technology, cluster computing and grid computing, peer-to-peer computing, network storage, and network security.

Shouhuai Xu received the PhD degree in computer science from Fudan University, China. He is an associate professor in the Department of Computer Science, University of Texas at San Antonio. His expertise and research interests include cryptography and mathematical modeling and analysis of cyber security.