2012 IEEE/ACM 45th International Symposium on Microarchitecture Workshops

Security Verification of Hardware-enabled Attestation Protocols

Tianwei Zhang Jakub Szefer Ruby B. Lee
Princeton University
{tianweiz,szefer,rblee}@princeton.edu

Abstract

Hardware-software security architectures can significantly improve the security provided to computer users. However, we are lacking a security verification methodology that can provide design-time verification of the security properties provided by such architectures. While verification of an entire hardware-software security architecture is very difficult today, this paper proposes a methodology for verifying essential aspects of the architecture. We use attestation protocols proposed by different hardware security architectures as examples of such essential aspects. Attestation is an important and interesting new requirement for having trust in a remote computer, e.g., in a cloud computing scenario. We use a finite-state model checker to model the system and the attackers, and check the security of the protocols against attacks. We provide new actionable heuristics for designing invariants that are validated by the model checker and thus used to detect potential attacks. The verification ensures that the invariants hold and the protocol is secure. Otherwise, the protocol design is updated on a failure and the verification is re-run.

1. Introduction

Although several papers have studied the design of hardware-enhanced security architectures, security verification of these architectures remains an unsolved problem. While functional verification of architectural designs exists, security verification has distinctly different needs. A systematic hardware-software security verification methodology is needed. We illustrate our security verification methodology with hardware-enhanced attestation protocols. Attestation is an essential aspect of establishing trust between a user and a remote computer, especially important in the era of cloud computing.

We describe attestation protocols for three classes of secure software-hardware architectures in Figure 1, based on three very different threat models. In the first class, both the hypervisor and the guest OS are trusted. In the second class, only the hypervisor is trusted, while in the third class, only the OS is trusted. In the first class, we use the remote attestation protocol defined for the Trusted Platform Module (TPM) [1]. In the second and third classes of architectures, we do not need a separate TPM chip but instead use processor-based attestation. New ideas such as a *layer-skipping trust chain* are used to enable more focused remote attestation reports for guiding decisions on whether specific tasks can run securely on remote systems, thus increasing the resilience of distributed systems. We show how to model and verify the attestation protocol used with an external TPM chip as a baseline. We then present streamlined protocols for the two new and interesting classes of untrusted OS and untrusted hypervisor architecture. We do not model the fourth class of architectures where both the hypervisor and the OS are untrusted, since no architecture has been proposed with that aggressive threat model yet.

In this paper we show how to model different components in a system and how to model attacks and protocols. We present a novel invariant selection heuristic. Then we use a finite-state model checking tool, Murphi [8, 21], to check the models of the protocols up to the invariants. The use of a model-checker can also help us improve the design of protocols as it reveals defects of the protocols and we can iteratively update them until all invariants pass. Writing verification models in a model-checker also allows us to compare the number of states explored and the number of rules fired for different protocols, as a rough gauge of the flexibility or complexity of the different attestation protocols.

Our methodology can be applied generally to verify protocols for distributed hardware-software systems. In this paper, we focus on designing and verifying attestation protocols. In particular, the startup attestation protocols are used to verify that correct virtual machines (VMs), or a set of trusted software modules, have securely started on a remote system. The startup attestation is crucial, as any later runtime attestations cannot be trusted if the initial state of the system was not verified to have been correct. Once the user has received the remote attestation, he can decide whether it is safe to execute his tasks remotely on the secure hardware.

The rest of the paper is organized as follows. Section 2 provides background on attestation under different threat models. Section 3 presents our verification methodology. Sections 4, 5 and 6 present the details of the attestation protocols for the three different classes of architectures. Section 7 discusses the evaluation of the verification, while section 8 lists related work. We conclude in Section 9.

2. Attestation

Attestation is an essential and interesting new requirement for having trust in a remote computer, e.g., in a cloud computing scenario. Attestation is often rooted in having a trusted hardware component which can be used to build trust in the rest of the system. In addition to co-processor based attestations such as using a TPM [25], there are processor-based attestations of Bastion [5, 4] and HyperWall [29], for example. These architectures can measure (usually through a cryptographic hash) trusted software modules needed for the safe execution of security-critical tasks, e.g. [5], or the whole protected VMs and their protections, e.g. [29]. The measurements are signed with a private key of the hardware, thus they are tied to the platform on which they were generated and they cannot be spoofed. Correct implementation of the architecture in turn guarantees that the measurements are made correctly.

2.1. Attestation under Different Threat Models

We use, as an example, attestation protocols designed for use by different hardware security architectures with different threat models. In Figure 1, we show a classification of different architectures based on two key characteristics. First, whether the guest OS inside the VM is trusted or not. Second, whether the hypervisor is trusted or not. These are important design points for a number of hardware security architectures, and picking one architecture from each design point provides us with a case study for verification under that threat model.

Systems using TPM-based attestation require having a full trust chain from BIOS and boot loader up to the attested software (as

978-0-7695-4920-0/12 $26.00 © 2012 IEEE
DOI 10.1109/MICROW.2012.16

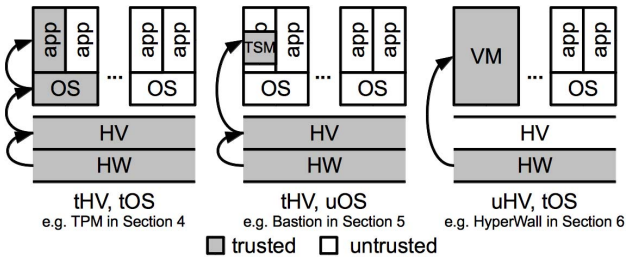


Figure 1: Different threat models: trusted or untrusted Hypervisor (tHV or uHV) and trusted or untrusted guest OS (tOS or uOS).

shown by the arrow chain in Figure 1). While TPM attestation can tell the customer whether any of these components has been compromised, a correct full chain is needed to attest any tasks running inside the VM. On the other hand, Bastion and HyperWall present a layer-skipping approach to attestation where parts of the chain are removed (since the removed components are not in the trust chain for confidentiality and integrity).

2.2. Protocols for Attestation Delivery

We present protocols for the delivery of the attestation measurements for each class of architectures. The differences in the protocols highlight the different threat models of the architectures and the trusted and untrusted components that we need to worry about. The TPM's protocol is based on past work, presented here as a baseline case due to TPM's wide use and study. The attestation protocols for Bastion and HyperWall are new protocols that we have improved from their authors' initial proposals. All three protocols deliver attestation measurements to a remote customer.

Upon receiving the attestation report, the user can check if the measurements (e.g., cryptographic hashes) of the VM or software modules correspond to the correct software and thus whether it is safe to execute his or her task remotely. Since the users are in possession of the programs and data they want to execute remotely, they can measure them locally to obtain a set of good measurements to compare against. The good measurements can also be available from a trusted source, as in the Integrity Measurement Architecture (IMA) [24], or the vendor of the software that the user is using. The checking of the measurement values is not part of the actual protocol, which only focuses on the secure delivery of the measured values.

3. Verification Methodology

We verify the protocols using a finite-state model checker. Automated finite-state verification is a robust field with many tools available (see related works in Section 8 for a sample list). However, not much work on the design and verification of protocols for hardware-software security architectures has been done. We aim to fill in this gap and present our methodology for protocol design and verification for such hardware-software security architectures.

3.1. Protocol Design

The design of protocols for startup attestation needs to answer four questions about the attested values.

When were the attestation measurements taken? This depends on the type of attestation. Our work focuses on attestation of the correct initialization of the VMs or software modules; hence the protocols require the attestation measurements to be taken at the initialization time of the VMs or the applications. For TPM, the attestation measurements are taken at bootup and hypervisor initialization.

Who took the attestation measurements? Only trusted components in the trusted computing base (TCB) should be allowed to take and share the measurements. For the architectures which assume the hypervisor or OS is trusted, these entities can take part in measurements generation; otherwise only the hardware components in the TCB can take part in generating the measurements.

Were the attestation measurements kept securely since generation? The protection of measurements rests with the architecture. This requires secure storage, for example, that ensures the protection of the measurements from the time they are taken until the time they are retrieved for attestation.

Are these measurements authentic and fresh? Authenticity of the measurements rests in the digital signature which binds the measurements to the underlying trusted hardware components of the system and such a signature must be included in the protocol. Use of a nonce provides freshness, so stale measurements cannot be replayed by an attacker at a later time.

3.2. System Modeling for Verification

To verify the designed protocols, we need to translate these protocols and their underlying architectures into representative yet tractable models. We take the steps below to build a model of the architecture, the attackers and the protocol, as a state machine. We then execute the state machine for model checking, to verify the protocol. A sample graphical specification of such a protocol is shown in Figure 2.

Subjects: The protocol modeling requires specifying subjects (vertical lines in Figure 2). Subjects are hardware, software or network components in the distributed system. A solid vertical line represents a trusted subject while a dashed vertical line represents an untrusted subject, which can be an attacker. Each subject has a set of states based on the protocol and each state has inputs and outputs. The transitions between different states are also defined by the protocol. For startup attestation protocols, the main subject is the customer, $CUST$, who asks for attestation of the remote system and receives attestation reports. Once the customer verifies the report and authenticates the system, it will reach the *commit state*. There is a network subject, NET, to represent the network through which the customer communicates with the remote system. And there are multiple subjects within the *Remote System* (e.g. hypervisor, OS, processor, etc.) depending on the individual architecture.

Attacks: Our proposed modeling does not attempt to enumerate various possible attacks and individual attackers. Rather, any values being relayed by an untrusted entity are modeled as outputs which can be in a "good" (correct value), "bad" (malicious value) or "old" (replayed value) state so the tool can model any modification or replay of values. Furthermore, any values derived by any subject from inputs that are in the "bad" or "old" state are said to be *tainted* and the taint is propagated to the final commit state where no tainted values should be accepted.

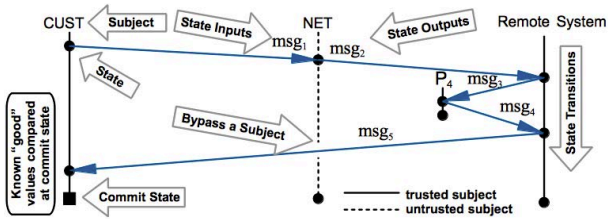


Figure 2: Graphical specification of a protocol model.

Invariants: Invariants are conditions which need to hold true in the *commit state* of our model. The invariants concisely capture the threats we are worried about and ensure that an attack has not happened. The verification is complete and the protocol deemed secure if no invariant is violated. Our heuristic for invariant selection is presented below.

Execution: Based on the above specification, a state machine is built that the model-checker can explore and verify. It starts in the initial state of the customer, *CUST*, and searches for all possible states. At each state, it takes actions corresponding to the rules. It will exhaustively explore all possible rules and states, to find all the possible paths from the initial state to the commit state. Then in these paths, the model-checker will judge if the invariants are satisfied. The protocol is verified to be secure if all invariants are satisfied in all paths from protocol initiation to the customer commit state. If any invariant fails, we fix the protocol and repeat the model-checking.

3.3. New Invariants Selection Heuristics

Invariant selection is a difficult problem. We take a step towards a systematic definition of invariants by specifying heuristics which have worked well for our own verification efforts. The focus of our heuristics is on invariants that need to hold true when the customer, *CUST*, accepts the attestation report and reaches the final commit state. The invariants are generated and evaluated in order, following the three rules:

1. Check for expected values – All received values should make sense given some value that was previously sent and remembered by the customer. For example, if a nonce was sent and later a hash over some values including the nonce is received then the hash needs to contain that nonce, otherwise the customer should not accept the message.
2. Check for self-consistency – All received values should be mutually consistent. For example, if message *M* is received with a hash of the message *hM* then the customer should not accept the message unless the *M* and *hM* match.
3. Check for tainted values – If a trusted subject receives a "bad" or "old" value as input from an untrusted subject, its output is tainted and the taint is propagated to the commit state. Any values derived from the tainted value later on are also tainted. At the commit state, inputs can be checked if they are tainted, tainted values should never be accepted in that state. For example, some values which are generated remotely have no known "good" values that the customer can check against (e.g. a remote VM generates a fresh key), i.e. there is no expected value to check against. Also, the self-consistency check may not catch this if a keyed hash is not used (most likely by design mistake). Hence tainting is needed.

3.4. Verification Workflow

Our general approach to building and checking a model of the system, the attackers and the protocol is:

1. Define the subjects which can see or affect the protocol and the information it passes; identify trusted and untrusted subjects;
2. Add states to each subject (whenever a subject sends output or accepts some input information related to the protocol, a state is needed to handle the communication);
3. Add outputs and inputs that are communicated between subjects at each of the subjects' states;
4. Model potential attacks: define all the bad or old variables output from the untrusted subjects.
5. Specify invariants for the protocol. Add tainting variables when a "bad" or "old" value of interest is output from an untrusted subject, so that state is captured and propagated to the *commit state*.
6. Run the model-checker and see if the invariants pass for every path through the model, from the protocol initiation state to the commit state.

If an invariant fails, the protocol needs to be fixed and the verification needs to be re-run. The process can be executed iteratively until no invariants cause a failure and require protocol improvements.

For this work we selected Murphi [8] as the model checking tool. Sections 4, 5 and 6 present attestation protocols for three classes of hardware-software security architectures. The invariants listed in these sections have been formulated using the above heuristics. Models are built and translated into Murphi code. The verification results are presented in each section. Section 7 summarizes the design and verification.

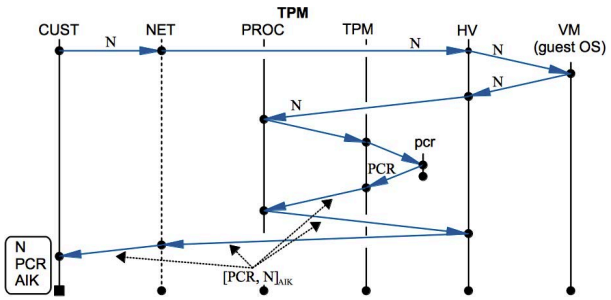
4. Case study: trusted hypervisor, trusted OS

We begin our design and evaluation of protocols for delivery of startup attestation with the class of architectures with a trusted hypervisor and a trusted guest OS (tHV, tOS). The Trusted Platform Module (TPM) [1] assumes such a TCB in the attestation that it provides. We use the TPM attestation protocol as a baseline protocol. Although many attestation protocols are built on TPM [25, 20, 3], few have focused on verification of the attestation protocol. We show how Murphi can be applied to verify this TPM attestation protocol, which is original work.We use an existing TPM attestation protocol as our baseline.

4.1. Required Architecture

TPM is a co-processor integrated into a hardware platform. It provides operations such as: hash functions (SHA-1 and HMAC), public-key encryption and decryption (RSA), signatures, random number generation, and it offers memory for persistent key storage. It protects against software attacks but not hardware attacks.

One of the main goals realized by the TPM is to perform attestation, i.e. measure the configuration of the system's software, and send this to a third party to verify that the software has not been changed.



(a) Protocol Diagram (tHV, tOS)

Cust.:	Generate nonce N
Cust. → Net → HV → VM:	Send N and "attestation request"
VM → HV → Proc. → TPM:	Send N and "attestation request"
TPM:	Calculate and sign PCR using AIK
TPM → Proc. → HV → Cust.:	Send signed attestation report
Cust.:	Decrypt the signature and verify attestation

(b) Protocol Notation (tHV, tOS)

Figure 3: Startup attestation protocol for the case of trusted hypervisor and trusted OS, e.g., for TPM.

The trusted computing base includes the entire hardware platform (including processor, memory and the TPM chip), the hypervisor and the guest OS in the VM. However, software applications in the VMs are untrusted. TPM's Platform Configuration Registers (PCRs) are used to store measurements about the platform configuration [12]. PCR values are initialized during the platform boot process. When software is loaded for execution, hash values of its code or configuration data are calculated and extended (i.e., accumulated) into the PCRs. A final PCR value is an accumulation of individual measurements and configuration data, which define the software as well as the order they are launched. It can be sent to the customer for bootup integrity attestation of the software. The PCR values are signed with the Attestation Identity Key (AIK). This key-pair is created by the TPM and certified by a trusted third party called the Privacy Certification Authority (Privacy CA). Different AIKs are distributed to different customers by the Privacy CA.

4.2. Description of Attestation Protocol

The protocol for delivery of TPM attestation reports is shown in Figure 3. The customer sends the request attestation to the TPM with a session nonce, N, which is used to prevent replay attacks. Then the TPM will retrieve the values of the PCR, sign these with the private AIK and send it back to the customer. When the customer receives the report, he can use the public AIK to decrypt the report. Comparison of the nonce inside is used to make sure that the attestation report is fresh, and then the PCR values can be checked to see if they match the expected values for the software components.

4.3. Protocol Invariants

We specify four invariants for the protocol according to the heuristics in Section 3.3. The invariants are listed below:

1. The signature received is fresh,
2. The attestation of requested PCRs returns the expected measurement values,
3. The customer's saved nonce was not tampered with, and
4. The PCR value used by the processor was not tampered with.

The nonce and PCR are known to the customer. So invariants 1 and 2 are attestation of *expected values*. The nonce and PCR are also checked to see if they are tainted, giving invariants 3 and 4. For easy comparison, Table 1 lists the invariants of all the architectures discussed in this paper.

4.4. Verification Results

Invariant (1) is satisfied because the nonce is used to ensure the freshness of the attestation. Invariant (2) passes because the customer and their local storage are assumed to be secure. So the nonce won't be leaked. Also, the Integrity Measurement Architecture (IMA) [24] can be used to get the list of the entities included in the measurement and expected good values of measurements for these entities. As a result, the customer is able to check if the received PCRs match the expected values, satisfying invariant (3). In this platform, the PCR and AIK are stored in the TPM, which is assumed to be secure. So invariant (4) is not violated.

5. Case Study: trusted hypervisor, untrusted OS

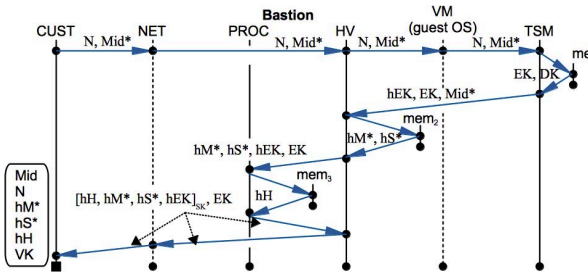
We now consider the class of trusted hypervisor and untrusted OS architectures (tHV, uOS), e.g., the Bastion [6] architecture.

5.1. Required Architecture

In this class, the aim is to protect Trusted Software Modules (TSMs) within an untrusted software stack. For example, in Bastion, the hardware first protects a trusted hypervisor. Upon hypervisor launch, the hardware computes a cryptographic hash over the identity of the hypervisor. This can be used for authentication of the hypervisor before it can access the secure storage area. The keys used to encrypt and fingerprint this cryptographically-secured hypervisor storage are securely stored in new processor registers. Next, the trusted hypervisor can now launch trusted software modules, using its hypervisor secure storage to store module states and "soft registers" for protecting these modules. The hypervisor computes the Module Identity (the hash over the module's initial code and data), and a hash over its Security Segment. The latter is a new data structure that includes the module's code, private data pages, authorized entry points and authorized shared memory interfaces. These values are stored in the hypervisor's secure storage area. For the protection of secure storage for the modules, the module will encrypt the data and save it to the disk. It will also calculate the hash of the encrypted data. Both the encryption key and the hash value are stored in the hypervisor's secure storage. Hence the data's confidentiality and integrity are protected by the security of the trusted hypervisor, while the hardware protects the hypervisor.

5.2. Description of Attestation Protocol

The protocol for delivery of attestation reports for Bastion [6] is shown in Figure 4. The customer sends the request for attestation to the module (TSM) with a session nonce, N, and a list of modules to attest, Mid^*, e.g. the target module depends on other modules that should be attested as well. Then the module generates an asymmetric key pair (EK and DK) to be used to establish a secure channel with the customer. The module stores the private key DK in an encrypted and integrity-checked memory page and creates a report containing



(a) Protocol Diagram (tHV, uOS)

Cust.:	Generate nonce N
Cust. → Net → HV → VM → TSM:	Send N, Mid* and "attestation request"
TSM:	Generate EK, DK and calculate hEK, the hash value of EK and N
TSM → mem1 → TSM:	Store EK, DK for later use
TSM → HV:	Send EK, Mid* and hEK
HV → mem2 → HV:	Look up hashes of Module Identity hM* and Secure Segment hS*
HV → Proc.:	Send hM*, hS*, hEK and EK
Proc. → mem3 → Proc.:	Look up hashes of Hypervisor Identity hH
Proc.:	Generate the signature with Proc's private key, SK
Proc. → HV → Cust.:	Send signed attestation report and EK
Cust.:	Verify attestation and save EK for secure communication with TSM

(b) Protocol Notation (tHV, uOS)

Figure 4: Startup attestation protocol for the case of trusted hypervisor and untrusted OS, e.g. for Bastion architecture.

the hash $hEK = h(EK, N)$ over the public key EK and nonce. It then invokes the hypervisor's attest routine to request hashes of the identities and security segments of the modules (hM^* and hS^*) to add to the report. The hypervisor compiles the report by looking up hash measurements for the modules listed in Mid^* and then asks the processor to add the measurement of the hypervisor hH to the report and sign it with the processor's private key SK. This is done by a new attest instruction in Bastion. The signature over the reported values, $[hH, hM^*, hS^*, hEK]_{SK}$, and the session's public encryption key EK are sent to the requester.

When the customer receives the report, first the processor's public key VK is used to verify the signature R on the attestation report. Then the known "good" values for the measurements of the modules and the hypervisor are compared with the received values. The customer should also calculate the hash value of received EK and N, compare it to the received hEK in the attestation report to check for self-consistency.

Improvements to Bastion protocol: Comparing with the protocol in [6], we remove the measurements of modules and hypervisor in the plaintext of the report because the signature is already sufficient to protect the integrity of these measurements. The customer should remember the module id he or she requested and has known "good" values of measurements to check if the reply contains the correct measurement values.

Moreover, the specification of the attest hypercall, which is part of the protocol, was not clear in the Bastion architecture [6] – if implemented wrongly it could have led to an attack. For instance, if the network or VM is untrusted, the attestation request could have been manipulated, allowing an untrusted module to masquerade as a good one. The untrusted module could generate its own set of keys and invoke a forged attestation hypercall to the hypervisor, but include the good module's id as if it came from the trusted module. Upon receiving the report, the customer would think he received the attestation from the correct module, and that the channel between him and the module is trusted.

To prevent this, the secure hypervisor must be aware of the module id of the module that sends the attestation hypercall and ensure that the attestation request includes that id in the list of modules to attest. Our new protocol uses the semantics of the fixed attest hypercall with this property.

5.3. Protocol Invariants

We specify seven invariants for the protocol according to the heuristics in Section 3.3. The invariants are listed below (and in Table 1, in column "tHV, uOS", for easy comparison with the other protocols):

1. The signature received is fresh,
2. The attestation of the requested Trusted Software Modules (TSMs) returns the expected measurement values,
3. The EK received was not tampered with,
4. The customer's saved nonce and module list requested for attestation were not tampered with,
5. The Hypervisor Identity hash stored in the processor was not tampered with,
6. Module Identity and Security Segment stored in hypervisor secure storage were not tampered with, and
7. TSM's memory and DK were not tampered with.

The nonce and TSM measurements need to be checked if they are as expected, giving invariants (1) and (2). The customer needs to check if the module's public key, EK, is self-consistent with the hash hEK, giving (3). The identity of hypervisor and modules as well as the TSM's private key need to be checked to see if they have been tainted, giving (5), (6) and (7).

5.4. Verification Results

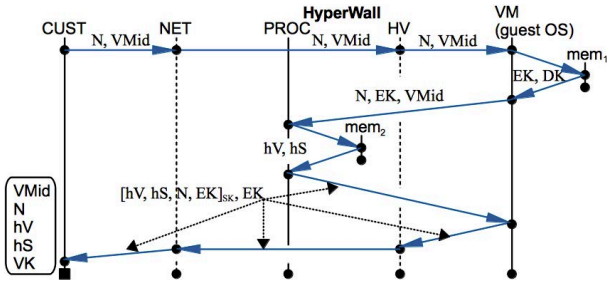
Using Murphi, we can prove that invariant (1) is never violated. This is because we use the nonce to ensure the attestation is fresh, so the replay attacks cannot happen in this protocol. Invariant (2) passes after Bastion specification is fixed (as described above) so that the attest hypercall checks the calling module's id. Invariant (3) passes as the trusted module will use the hash function to protect the integrity of its decryption key. Any potential attacks to tamper with EK will be detected by the customer by regenerating the hash of (EK, N) and comparing with hEK Invariant (4) passes because any modifications by the untrusted subjects along the way, like the network and the guest OS, is detected due to signature and hashes. In addition, Bastion ensures the secure storage of Module Identities and Module Security Segments, as well as Hypervisor Identity, so invariants (5) and (6) pass. In Bastion architecture, the decryption key DK is stored in the secure storage of the trusted software module, so invariant (7) passes.

6. Case Study: untrusted hypervisor, trusted OS

We now consider the class of untrusted hypervisor and trusted OS (uHV, tOS). While a trusted hypervisor can provide many useful features, a number of researchers have begun to explore alternative designs of a virtualized environment with an untrusted hypervisor [28, 29, 14]. This reduces the systems' trusted computing base.

6.1. Required Architecture

In this class of architecture, the hypervisor is removed from the TCB and the goal is to protect guest VMs from a compromised hypervisor. To realize this in the HyperWall architecture [29], a new Confidentiality and Integrity Protection (CIP) table is introduced. The CIP table is implemented in a special portion of DRAM and accessible only from



(a) Protocol Diagram (uHV, tOS)

Cust.:	Generate nonce N
Cust. → HV → VM:	Send N, VMid and "attestation request"
VM:	Generate EK, DK
VM → mem1 → VM:	Store EK, DK for later use
VM → Proc.:	Send EK, VMid and N
Proc. → mem2 → Proc.:	Look up hashes of VM initial state hV and protection hS
Proc.:	Generate the signature signed by SK
Proc. → VM → HV → Cust.:	Send signed attestation report and EK
Cust.:	Verify attestation and save EK for secure communication with VM

(b) Protocol Notation (uHV, tOS)

Figure 5: Startup attestation protocol for the case of untrusted hypervisor and trusted OS, e.g. for HyperWall architecture.

the trusted processor hardware's memory management unit (MMU) and input/output memory management unit (I/O MMU). The CIP table stores mappings of hypervisor and DMA access rights to the machine memory pages assigned to VMs. Because of these mechanisms, a malicious hypervisor cannot access the VM's memory directly (or through DMA by configuring a device) and the confidentiality and integrity of the VMs are protected.

When a VM is started, the CIP hardware updates CIP tables with the protections requested for that VM by the customer. Once successfully initialized, the hardware makes sure that the hypervisor (or devices via direct memory access, DMA) is not able to access the protected memory regions. The hardware generates the hash of the initial CIP protections and the hash of the initial memory contents, and updates the attestation measurement signature. The hypervisor sends the signature back to the customer to attest the virtual machine that was started.

Because portions of the VM's code and data memory are now protected from hypervisor or DMA access, the VM can generate and store secrets in this memory. Specifically, the VM may generate a public-private key pair (EK, DK) and store it in the protected memory. So the VM can communicate with the customer securely, preventing attacks on its confidentiality and integrity from the malicious hypervisor. The keys (EK, DK) are tied to the hardware through the `sign_bytes` instruction [29]. The hardware has the public-private key pair (VK, SK) and uses the SK signing key when the `sign_bytes` instruction is invoked. This instruction does not trap to the hypervisor, so the key cannot be intercepted. The SK is unique to each microprocessor implementing the HyperWall architecture and the customer can obtain certificates from the cloud provider for the public processor key VK corresponding to SK on the machine where the VM is executing, to verify the signature. The hardware key vouches for the soft keys (VK, SK) generated by the VM and the customer knows that the VM which sends him the EK is the one executing on (and protected by) a HyperWall machine.

6.2. Description of Attestation Protocol

The protocol for delivery of attestation reports for HyperWall is shown in Figure 5. First, the customer requests attestation of the VM $VMid$, and specifies a nonce N. This is communicated directly to the VM and sent in clear text (although communication between the VM and customer can be secured with the SSL protocol to prevent third parties from eavesdropping, the target VM has not been attested yet, so it has to assume the information is visible to others).

Next, the VM generates a public-private key-pair (EK, DK). EK, the nonce and $VMid$ are passed along to the hardware for attestation. The processor retrieves the hash measurements of the initial state hv and protections hs of the VM, and generates a signature of the request using its SK key. These are relayed by hypervisor and VM back to the customer.

Improvements to HyperWall Protocol: HyperWall originally used two protocols to archive what we can do in one protocol: The first one was a challenge-response protocol for the VM startup; the second one was a modified SSL protocol for secure communication establishment. Since both the VM's protection and its public key need the certification of the processor, we can easily put all these values in a single signature, and realize the two goals in one procedure. Moreover, the public key of the VM can then be used with a standard SSL protocol. These improvements were made using our verification methodology.

6.3. Protocol Invariants

We specify six invariants for the protocol according to the heuristics in Section 3.3. The invariants are listed below (and in Table 1, in column "uHV, tOS", for easy comparison with the other protocols):

1. The signature received is fresh,
2. The attestation of the requested VM returns the expected measurement values,
3. EK received was not tampered with,
4. The customer's saved nonce and $VMid$ were not tampered with,
5. VM image hash and protections hash used by the processor in attestation were not tampered with, and
6. VM's memory and DK were not tampered with.

The customer needs to check the expected values of nonce and VM measurements, giving invariants (1) (2). The customer should do the self-consistency check for the public key sent, EK, with the hash of it that is also sent, giving (3). The nonce, VM's measurements, identity, protections and private keys should not be modified by an untrusted party, i.e., not tainted, giving invariants (4), (5) and (6).

6.4. Verification Results

The nonce N needs to be included in the final reply message to the customer, passing invariant (1). While the hypervisor could modify the nonce, $VMid$ or EK as the attestation request passes through it, the attestation request from the VM goes directly to the hardware, bypassing the hypervisor, ensuring invariant (2) passes. This also ensures the nonce and EK are not tampered with. Inclusion of EK in the reply message ensures that the customer is able to recognize a potentially compromised EK (e.g. compare the received EK to the value inside the signature, using the known public key of the processor) and invariant (3) passes. Invariant (4) passes because any modification by the untrusted subjects along the way, like the network, is detected due to signature and hashes. To pass invariant (5), the processor needs to have some memory which cannot be tampered with, ensuring that hashes of the initial VM image or protections are not compromised. The HyperWall architecture has memory accessible only to hardware, thus passing invariant (5). To ensure that DK or the VM's protected code or data are not exposed, VM needs protected memory that will not be accessible to the hypervisor or other VMs. HyperWall's CIP protections for VM memory enable passing invariant (6).

7. Evaluation

7.1. Verification Complexity

Thanks to the use of Murphi, we can quantitatively compare the models of the attestation protocols. Table 2 attempts to summarize the modeling and verification complexity for each model.

The Murphi states explored are a function of the number of subjects and their states. The Murphi rules fired are a function of the number of variables passed between each subject's states. The number of invariants is proportional to the number of untrusted subjects modeled and the number of values checked at the commit state. Lines of verification files includes comments and commented-out code, which are crucial for clarifying modifications made when ensuring that the invariants pass, so this is reported rather than source lines of code (SLOC). Model execution time is the run time of verification of each model on a commodity Dell R610 system with Linux OS and CMurphi 5.4.4.

One interesting pattern is that a protocol's and architecture's flexibility will usually show up as a higher number of Murphi states and

Table 1: Summary of invariants for the protocols for the three types of architectures, invariant types were explained in Section 3.3.

Invariant Type	tHV, tOS (e.g. TPM)	tHW, uOS (e.g. Bastion)	uHV, tOS (e.g. HyperWall)
expected values	1. Received signature is fresh	1. Received signature is fresh	1. Received signature is fresh
	2. Received the expected measurements of PCRs	2. Receiving the expected measurements of the TSMs requested	2.Receiving the expected measurements of the VM requested
self-consistency	—	3. EK received was not tampered with	3.EK received is not tampered with
tainted values	3. Nonce was not tampered with	4. Nonce and modules list were not tampered with	4.Nonce and $VMid$ were not tampered with
	4. PCR values were not tampered with	5. Hypervisor Identity was not tampered with	5.The hashes of the VM image and its protections were not tampered with
	—	6. Module Identity and Secure Segment hashes were not tampered with	—
	—	7. TSM's private key DK was not tampered with	6.VM's private key DK was not tampered with

Table 2: Summary of Murphi verification of the protocols.

Architecture	Mur -phi States	Murphi Rules Fired	In -var -iants	Lines of file	Model run time
tHV, tOS e.g. TPM	720	7236	4	722	0.53s
tHV, uOS e.g. Bastion	7920	265392	7	963	0.61s
uHV, tOS e.g. HyperWall	1584	39852	6	829	0.56s

rules fired. In Bastion, for example, multiple software modules can be attested in one run of the attestation protocol. To achieve this, however, Bastion requires retrieving measurements from multiple secure memory locations (modeled as mem_2 and mem_3 subjects in Figure 4). This is reflected in higher numbers in Table 2. Also, while the TPM attestation only reports the software loaded and measured at boot-time, the Bastion and Hyperwall initialization protocols report the secure launching of arbitrary trusted software modules or Virtual Machines, respectively, and the creation of a public-private key pair for future secure communications with the customer.

7.2. Security Discussion

First, TPM, Bastion and HyperWall all help to improve the integrity of certain trusted software components. TPM aims to protect the integrity of the entire software stack. Bastion tries to protect the secure launch of a set of trusted software modules in a VM by reporting the measurement of the identity and security segment of each of these modules, and the measurement of the trusted hypervisor. HyperWall aims to protect the VM's secure initialization by reporting the measurement of the VM's image and requested protections of the VM. All these values can be sent to the remote customer in a signed attestation report. Our verifications assure the correctness of these startup attestation protocols.

Second, we build a secure communication channel between the remote customer and the trusted software components in the last two architectures. The customer can talk securely to TSMs in Bastion, and to VMs in HyperWall. Our verification shows that these communication channels are protected and the untrusted components in each architecture have no access to the newly generated private key.

We note that the security verification of the protocols depends on trusted components remaining trusted. For example, in Figure 3 for the TPM attestation protocol, the entire hardware platform (including the processor, the TPM chip and the main memory), the hypervisor and the Virtual Machine (including the guest OS and the application) are all assumed to be trusted components. If any of these trusted components is compromised, then the security verification of TPM's attestation protocol becomes null and void.

8. Related Work

Our work is motivated by many proposals for hardware security architectures, e.g. [27, 19, 6, 29, 14, 17, 9], commercial solutions, e.g. [1, 10] and the fact that few of these are verified for security properties.

Closely related to our work, XOM [18] and SecVisor [26] have received the benefit of verification using model-checking. XOM verification checked read-protection and tamper-resistance against a simplified model of the instruction set operations on registers and memory. SecVisor verification [11] used a logical system to reason that the security demonstrated by a small model of the SecVisor reference monitor can scale to the implementation size. Both verification efforts found defects in the original designs.

Our work uses automated finite-state verification, which is a robust field: security verification has been performed with many tools aside from Murphi, e.g PRISM [16] or Alloy [13]. This approach has been used to verify a large number of diverse systems, including network security protocols like SSL [22], multiprocessor memory consistency protocols [23], linux file systems [30], web security techniques [2] and browser-based isolation [7]. Machine-aided proof techniques have also been able to verify the security of an OS microkernel [15].

In contrast to the above, our work presents two new protocols for delivery of startup attestation and is the first to look at applying the finite-state model checking to these protocols for hardware-software security architectures.

9. Conclusion

In this work we presented modeling and verification of protocols for delivery of startup attestation for three different classes of security architectures, where the OS or hypervisor can be untrusted. The TPM attestation protocol was presented as a baseline for comparison with the processor-based attestation protocols. The streamlined definitions of the startup attestation protocols, with an untrusted OS or untrusted hypervisor, are new contributions in the paper. We also detailed a methodology of modelling the protocols, the underlying system and attackers, and verifying the protocols using the Murphi finite-state enumeration tool. We also defined a methodology and new heuristics for the invariants generation. These heuristics can be used to generate

the invariants to improve and verify the protocols. We hope this work will encourage more tools and methodologies needed for systematic, design-time verification of hardware-enhanced security architectures.

References

[1] "Trusted Computing Group. TCG TPM Specification. http://www.trustedcomputinggroup.org/."

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a Formal Foundation of Web Security," in *Proc. of the Computer Security Foundations Symposium*, ser. CSF, July 2010.

[3] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proc. of the Conference on Computer and Communications Security*, ser. CCS, Oct. 2004.

[4] D. Champagne, "Scalable security architecture for trusted software," Ph.D. dissertation, Princeton University, Electrical Engineering Department, Princeton, NJ, 2010.

[5] D. Champagne and R. B. Lee, "Processor-based tailored attestation," *Princeton University Department of Electrical Engineering Technical Report*, Nov. 2010.

[6] D. Champagne and R. Lee, "Scalable architectural support for trusted software," in *Proc. of the Int. Symposium on High Performance Computer Architecture*, ser. HPCA, Jan. 2010.

[7] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: get the security of multiple browsers with just one," in *Proc. of the Conf. on Computer and Communications Security*, ser. CCS, Oct. 2011.

[8] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *Proc. of the Int. Conference on Computer Design: VLSI in Computer & Processors*, ser. ICCD, Oct. 1992.

[9] J. S. Dwoskin and R. B. Lee, "Hardware-rooted trust for secure key management and transient trust," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 389–400. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315294

[10] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. Smith, "Building the IBM 4758 secure coprocessor," *IEEE Computer*, Oct. 2001.

[11] J. Franklin, S. Chaki, A. Datta, and A. Seshadri, "Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size," in *Proc. of the Symposium on Security and Privacy*, ser. S&P, May 2010.

[12] T. C. Group, *TCG PC Specific Implementation Specification*, 2003.

[13] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[14] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural Support for Secure Virtualization under a Vulnerable Hypervisor," in *Proc. of Int. Symposium on Microarchitecture*, ser. MICRO, Dec. 2011.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proc. of the Symposium on Operating Systems Principles*, ser. SOSP, Oct. 2009.

[16] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. of the Int. Conference on Computer Aided Verification*, ser. CAV, July 2011.

[17] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 2–13. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2005.14

[18] D. Lie, J. Mitchell, C. Thekkath, and M. Horowitz, "Specifying and verifying hardware for tamper-resistant software," in *Proc. of the Symposium on Security and Privacy*, ser. S&P, May 2003.

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *SIGPLAN Notices*, Nov. 2000.

[20] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Proc. of the Symposium on Security and Privacy*, ser. S&P, May 2010.

[21] J. C. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using murphi." IEEE Computer Society Press, 1997.

[22] J. C. Mitchell, V. Shmatikov, and U. Stern, "Finite-state analysis of ssl 3.0," in *Proc. of the USENIX Security Symposium*, Jan. 1998.

[23] S. Qadeer, "Verifying sequential consistency on shared-memory multiprocessors by model checking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 8, pp. 730–741, Aug. 2003.

[24] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *Proc. of the USENIX Security Symposium*, Aug. 2004.

[25] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Journal of Science of Computer Programming*, vol. 74, no. 1-2, Dec. 2008.

[26] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proc. of the Symposium on Operating Systems Principles*, ser. SOSP, Oct. 2007.

[27] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *Proc. of the Int. Conference on Supercomputing*, ser. ICS, June 2003.

[28] J. Szefer and R. Lee, "A case for hardware protection of guest vms from compromised hypervisors in cloud computing," in *Proc. of the Int. Conf. on Dist. Computing Systems Workshops (ICDCSW)*, June 2011.

[29] J. Szefer and R. B. Lee, "Architectural Support for Hypervisor-Secure Virtualization," in *Proc. of Int. Conf. on Architectural Support for Prog. Languages and Operating Systems*, ser. ASPLOS, March 2012.

[30] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," *ACM Trans. Comput. Syst.*, Nov. 2006.