Practical and Scalable Security Verification of Secure Architectures

Tianwei Zhang
Nanyang Technological University
Singapore
tianwei.zhang@ntu.edu.sg

Jakub Szefer
Yale University
USA
jakub.szefer@yale.edu

Ruby B. Lee
Princeton University
USA
rblee@princeton.edu

ABSTRACT

We present a new and practical framework for security verification of secure architectures. Specifically, we break the verification task into external verification and internal verification. External verification considers the external protocols, i.e. interactions between users, compute servers, network entities, etc. Meanwhile, internal verification considers the interactions between hardware and software components within each server. This verification framework is general-purpose and can be applied to a stand-alone server, or a large-scale distributed system. We evaluate our verification method on the CloudMonatt and HyperWall architectures as examples.

ACM Reference Format:
Tianwei Zhang, Jakub Szefer, and Ruby B. Lee. 2021. Practical and Scalable Security Verification of Secure Architectures. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP '21), October 18, 2021, Virtual, CT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3505253.3505256

1 INTRODUCTION

Over the last decade, a number of secure architectures have been designed to provide security functionalities (e.g., XOM [19], AEGIS [30], SP [17], Bastion [7], HyperWall [33], DataSafe [8], Sanctum [10], or HDFI [28]). Ideas presented by some of these architectures have been implemented in commercial designs, such as ARM TrustZone [34], Intel's SGX [21], AMD's SEV [2].

Once any such security architecture is designed, it is necessary to check that there are no security vulnerabilities with the design that could allow an attacker to subvert the protections. Unlike software-based solutions which may be easily patched in the field, hardware architecture protections need to be correct from the beginning, as it is expensive and often not possible to update or replace them once hardware is manufactured. To address this issue, designers run extensive tests and simulations to make sure that the mechanisms work correctly. Moreover, the designers perform informal security evaluation which attempts to qualitatively reason about potential attacks and show how the architectural mechanisms prevent them. There is a lack, however, of a systematic methodology for verification of security architectures that can be applied to any architecture or system in a scalable manner.

A big challenge in verifying secure architectures is that secure architectures today are usually very complex. A secure architecture is likely to consist of different types of computing servers, and the end users. All of these are connected by networks. Meanwhile each computing server consists of different layers of software and hardware components. The secure operations of the architecture include the mutual communication between servers and users across the networks, as well as interactions between hardware and software modules inside a server. So verification of complex architectures thus needs to be achieved by focusing on two key aspects: external protocols and internal interactions.

Our contribution in this paper is the definition of a general-purpose security verification framework. It has different advantages. First, it is scalable to verify complex secure architectures. The presented approach breaks down a secure architecture into smaller components for verification. Specifically, verification of a secure architecture can be achieved effective by focusing on external verification, of the external protocols, and internal verification, of the internal interactions. External protocols are used for communication between servers and users, while internal interactions are for interactions among components within each server. We build state machines to verify the external protocols and internal interactions, thus effectively achieving verification scalability.

Second, our methodology is general-purpose and can be applied to different architectures. This method is not restricted to specific tools: designers can choose the tools they prefer to do the verification following our methodology. This achieves great practicality and granularity. We provide two case studies: verifying CloudMonatt [36] using a cryptographic protocol verifier ProVerif [4], and verifying HyperWall [32, 33] using a generic model checker Murphi [14]. These case studies show that our solution has been partly used to help design and enhance the secure architectures.

In summary, our contributions are:

- A new, general-purpose security verification framework for secure architectures and systems.
- A methodology to break the verification task of secure architectures and systems into external and internal verification, which can also be done hierarchically.
- A method to model different entities and components of such architectures as finite state machines.
- Evaluation of the methodology on different architectures using different tools.

HASP '21, October 18, 2021, Virtual, CT, USA

ACM ISBN 978-1-4503-9614-1/21/10…$15.00
https://doi.org/10.1145/3505253.3505256

We introduce our verification methodology and framework in Section 2. Using this methodology, we verify two secure architectures as case studies in Sections 3 and 4. We show the verification performance in Section 5. We summarize related work in Section 6 and conclude in Section 7.

2 VERIFICATION APPROACH

The security verification of secure architectures goes beyond functional verification. During design time, the threat model is specified, which lists the potential attackers and their capabilities. The security verification methodology needs to model enough aspects of an architecture to capture all possible behaviors of these attackers with their capabilities, and to model their impacts on the architecture.

A secure architecture usually consists of different components (e.g., distributed nodes, software and hardware modules). The interactions between these components and with the external entities (e.g., remote users, networks) are very complex [7, 36]. To achieve the scalability of verification, it is necessary that the verification is done on each part of the architecture, rather than on the whole architecture at once. Still, the verification of the sub-parts must compose into the verification of the whole architecture.

In Section 2.1 we propose a method of breaking the security verification of a system into smaller tasks, i.e., external verification and internal verification. In Section 2.2 we describe the detailed steps to conduct each verification task.

2.1 External and Internal Verification

A system is composed of many components. Each component is realized by one or more mechanisms. We specify *external protocols* as the interaction of the system with distributed or remote components, e.g. remote users, network, etc. There are also *internal interactions* which are interactions between components within a physical server or local system, e.g. processor, hypervisor, OS, etc.

The important aspect of the security-critical external protocols and internal interactions is that these involve untrusted principals or components, and hence involve potential attacks that we need to check for. This has led us to the proposition that the components' interactions are the most important parts to verify when considering the security of the system [7, 36–39]. By focusing on the component interactions we have found a natural breakdown of the architecture into smaller parts. Verifying smaller parts helps us avoid the state explosion problem.

The security verification of the external protocols and internal interactions provides coverage of more of the system because the focus is on how components interface with each other, and the details of the mechanisms are abstracted away. A component, even a whole server, can be treated as a blackbox during external verification – and in turn security verified during internal verification steps.

Identifying protocols and interactions. To find the different security-sensitive interactions, we identify different execution phases of a secure architecture or system, as shown in Figure 1. The middle six phases will be repeated many times during system runtime, while the other two phases correspond to system startup and shutdown. Each of the phases will have an external protocol if there is communication with the end user during that phase, and one or more internal interactions. The internal interactions occur when there is an event that will cause security-related state to be altered inside the trusted components of the architecture. The different execution stages of a hardware secure processor architecture shown in Figure 1 can be used to help identify protocols and interactions for security verification.

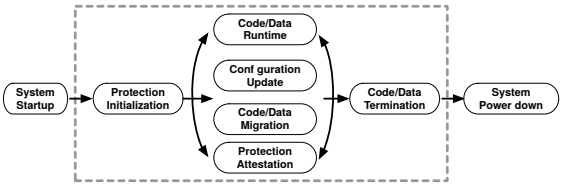

Figure 1: Different execution phases of a secure architecture.

Secure composition. Given secure mechanisms or protocols, *A* and *B*, which have been verified, it is very difficult to prove that the composition of the two is also secure. We do not tackle the problem of formal proofs of composability in this work. We focus on providing a sound methodology for the practical and scalable security verification of individual external protocols and internal interactions.

However, Protocol Composition Logic (PCL) has been proposed [13]. The composition theorems in PCL allow proofs of complex protocols to be built up from proofs of their constituent sub-protocols. It may be possible to build on such existing work as PCL to check the composition of the protocols and interactions which we verify.

2.2 Security Verification Framework

To verify a system's protocols and operations, we first build models for the system, and identify the trusted and untrusted subjects in the system. We specify the verification goals and invariants based on the system's functionality. Then we implement the models and test through the system models. If an invariant fails in some cases, a vulnerability has been found and the design needs to be updated.

2.2.1 Modeling System. **Specifying essential components.** A designer has to enumerate the components or principals involved. For the external protocol, we treat a physical server as one component. The network component, customers, cloud provider, and (if needed) trusted third party are also explicitly included for the benefit of the external protocols which involve the remote customer connecting via a communication path to the server. For the internal interactions, we consider the hardware components (e.g., microprocessor, memory chips, co-processors) and software components (e.g., applications, hypervisor, OSes). Among the protocol or interaction participants, there are untrusted components or principals, which could be sources of potential attacks. The untrusted components or principals are the potential attackers and their capabilities need to be checked.

Symbolic modeling. We adopted the symbolic modeling method [5], where the cryptographic primitives are represented by function symbols and perfect cryptography is assumed. Each component's operation can be represented as states of a state machine, and communication among the components can be represented as messages sent between the components. So we model each component as a

subject. Each subject has a set of states with inputs and outputs based on the system operation. The transitions between different states are also defined by the architecture designs and protocols.

Among all the subjects, there is an *initiator subject* that starts the system protocol/interaction and a *finisher subject* that ends the protocol/interaction; they could both be the same subject. This initiator subject has a *"start" state* while the finisher subject has a *"commit" state*. The verification procedure starts at the initiator's "start" state. At each state in each subject, it takes actions corresponding to the transition rules. It will exhaustively explore all possible rules and states to find all the possible paths from the initiator's "start" state to the finisher's "commit" state. Then we judge if the verification goals are satisfied in all of these paths. The system is verified to be secure if *there are paths from initiator's "start" state to finisher's "commit" state, and all the verification goals are satisfied in any of these paths.*

2.2.2 Preconditions and Security Invariants. The protocols and interactions are subject to constraints, the so-called preconditions. Preconditions are closely related to the trusted computing base (TCB) and often reflect which principals need to be in the TCB. If a precondition is removed, the protocol or interaction may no longer be verifiable. Ideally, during verification of a system, the minimal number of preconditions is determined, which can reduce the size of the Trusted Computing Base (TCB). One key benefit of our methodology is that it allows preconditions to be removed, (even though initially thought to be required), as verification passes with these preconditions removed.

Each protocol or interaction needs to satisfy certain security invariants – these invariants are only verified if for all possible execution traces, the invariant is not found to be violated. Thorough analysis of the protocols allow us to define the invariants correctly. Often the invariant is the goal of the design so correctness is clear. The security invariants focus typically on confidentiality and integrity of sensitive information. In the case of secure architectures, this sensitive information typically is: code or data executed or stored on the system, and measurements of the state of the system.

Confidentiality Validation. Each principal has access to various values, including ones tagged as confidential to indicate the need for confidentiality protection of that value. The untrusted principal could try to combine all the information it has obtained in all of its states to try to break confidentiality of some of the messages (e.g. it has seen cipher text in some state, and the decryption key in another).

For each value tagged as confidential, the invariants check if any untrusted principal has access to it. If not, confidentiality of this value is maintained. Otherwise the invariants check if the value is tagged as encrypted (i.e. it has a decryption key associated with it) and the untrusted principal has access to the key. If so the untrusted principal can obtain the plaintext, thus violating confidentiality. Otherwise the confidentiality is preserved. The above heuristics are consistent with our assumption of strong cryptography and that the attacker is not able to break the asymmetric or symmetric key cryptography, unless they have access to the proper key.

Integrity Validation. The way we are able to check for integrity attacks is through comparing the values available to an individual trusted principal to all the values in the model. The trusted principals have only visibility into their input values and the known-good `private` values they posses. Meanwhile, the model has visibility into all the inputs and outputs from all the principals, and which other principals may have modified these values. During a run of the model, the invariants check if there is enough information in the (explicit and implicit) inputs to a trusted principal for that principal to reject any inputs that have been compromised (e.g. fabricated or replayed values). The key ideas behind the integrity checks are: (1) checking for "known-good" values, which can be referenced by a trusted party to validate some of the inputs, these good values need to be stored securely or come from a trusted source; (2) checking for self-consistency of values, which allows a trusted party to check the inputs and make sure they are mutually consistent.

2.2.3 Implementation and Results. Our security verification methodology can be realized using very different tools. Since these are existing tools, the incremental overhead to achieve our security verification methodology is very small. Also, designers can choose the tools they are more familiar with, or that best suit their purpose. In this paper we use two verification tools ProVerif [4] and Murphi [14] to exemplify that this is a flexible methodology. Proverif has built-in security invariant checking support which Murphi does not. But Proverif is targeted at network protocol verification, and we have to use (repurpose) it in a clever way for checking interactions between software and hardware modules within a system. Murphi has more complete model checking facilities which enable the designer to do functional modelling and verification with the same tool as security verification. Murphi can be enhanced with security checking mechanisms as we have done, to propagate security tags for checking for integrity and confidentiality breaches. Designers can choose other model checker tools as well, e.g., Event-B [1], PAT [31], Scyther [12], Tamarin [24].

The verification results are either 1) the protocol or interaction passes, or 2) there is some invariant that does not hold and verification fails. If verification fails, the design needs to be updated, and one has to run the verification process again. When verification passes, some preconditions can be removed to test if they are necessary. Once the protocol or interaction passes with the least number of preconditions, the verification process is completed.

In the following two sections we validate our methodology on two types of secure architectures: a standalone server processor (HyperWall [32, 33]). and a distributed cloud system (CloudMonatt [36]).

3 VERIFYING A STANDALONE SERVER

In this section, we show how to use the above methodology to verify a secure standalone server processor. We use HyperWall [32, 33] as an example.

HyperWall is a secure processor architecture which aims to protect virtual machines from an untrusted hypervisor, a predecessor to AMD's SEV extensions. The processor hardware in HyperWall is extended with new mechanisms for managing the memory translation and memory update so that the hypervisor is not able to compromise confidentiality and integrity of a virtual machine. The hardware allows the hypervisor to manage the memory, but once the memory is assigned to a virtual machine, the hypervisor has

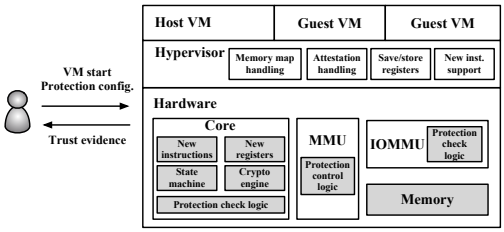

Figure 2: Architecture of HyperWall

no access to it. It is scrubbed by hardware before the hypervisor can gain access again. These protections are realized in HyperWall through extra registers and memory regions which are only accessible to the hardware, namely the TEC (Trust Evidence and Configuration) memory region. The TEC tables protect the memory of the guest VMs from accesses by the hypervisor and/or by DMA, depending on the customer's specification. Each memory region has an associated entry in the TEC tables specifying the access rights.

HyperWall can be used as the cloud server in a cloud computing scenario where there is a remote user communicating to his or her (HyperWall) server located in the cloud possibly managed by an untrusted cloud provider. HyperWall architecture is summarized in Figure 2. Below we present verification of one external protocol and one internal interaction of HyperWall. We have further performed verification of five more HyperWall interactions, summarized in Section 5.

3.1 External Protocol: VM Startup Validation

The security verification goal is to check if the integrity of VM image and configurations are protected during VM startup (system startup phase in Figure 1).

Modeling. Figure 3 shows the external protocol with the involved components. The customer component "starts" VMs by specifying a nonce, **N**, the virtual machine image **I**, and the desired set of confidentiality and integrity protections for the virtual machine, **P**. This "start VM" message is sent over the network to the hypervisor, which creates a data structure representing a VM. The network and hypervisor are both untrusted and have the same attack capabilities; thus we collapse them into one component for the purpose of modeling. After the VM is prepared, the processor is invoked to start the VM, through a VM Launch instruction. The microprocessor hardware launches the VM. It signs – with its secret key $\mathbf{SK_P}$ – values that will define the VM: **N**, **VID** (the VM identifier assigned by the processor), hash(**I**), hash(**P**), and **TE** (the initial trust evidence where initially the number of memory access violation is zero). The five values and their signature, **Sig**, and a certificate from the hardware manufacturer with the verification key needed to check the signature, $\mathbf{Cert_{VK_P}}$, are sent back to the customer. $\mathbf{Cert_{VK_P}}$ is signed by the trusted vendor.

To aid the verification, we have added two extra states to make explicit information available to the customer and processor. In particular, the customer knows the certificate for the manufacturer $\mathbf{Cert_{Mfg}}$ and the initial expected value of **TE**. The processor knows the key, $\mathbf{SK_P}$ that it uses to make the signatures. It also has a certificate for the corresponding public key, $\mathbf{VK_P}$, for recipients to

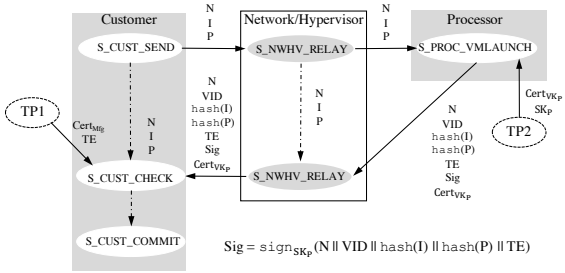

Figure 3: Model of VM Startup Validation external protocol. $\mathbf{SK_P}$ is the private key belonging to the processor, for which the customer has $\mathbf{Cert_{Mfg}}$ certificate from manufacturer and the $\mathbf{Cert_{VK_P}}$ certificate of the $\mathbf{SK_P}$ sent by the processor, which is signed by the manufacturer.

verify its signatures. This information is made explicit as inputs from the two trusted party states, TP1 and TP2.

Security invariants. We identify one invariant:

① The customer is able to reach the commit state with **N**, **VID**, hash(**I**), hash(**P**), **TE**, **Sig** and $\mathbf{Cert_{VK_P}}$ not being compromised by the untrusted hypervisor or the untrusted network.

Preconditions. We make several preconditions about the processor and cloud user and check if the above security invariants can be satisfied with these.

(C1) The processor is trusted.
(C2) The processor has valid $\mathbf{Cert_{VK_P}}$ and $\mathbf{SK_P}$.
(C3) The customer has valid $\mathbf{Cert_{Mfg}}$ and **TE**.

Implementation. We model the Customer, Network, and Processor in Murphi as a set of state machines. For this protocol, we are concerned with the network or hypervisor component fabricating or replaying values as it passes them to the processor, or when it returns values back to the customer. These two are collapsed into the single untrusted principal with states corresponding to two points where this principal needs to relay the data and it could be attacked.

We extend the murphi model checker tool to propagate multiple values, for each value whose integrity must be verified: the correct value, a fabricated value and a replayed value. At the commit state, we check if the cryptography used allowed us to verify that the correct value was returned, despite transmission through the untrusted network and hypervisor.

Results. The security verification passes for all possible runs and the customer can reach the commit state. The integrity of **N**, **VID**, hash(**I**), hash(**P**), **TE**, **Sig** and $\mathbf{Cert_{VK_P}}$ is protected against fabrication of values and replay of values.

Specifically, **N** and **TE** satisfy the case that there are known good values to compare against for these invariants. For $\mathbf{Cert_{VK_P}}$ there is the $\mathbf{Cert_{Mfg}}$ that can be used to compare against it and verify it. **VID** satisfies the case that there is a signature that includes this value and a chain of certificates to verify the verification key of the signature. hash(**I**) and hash(**P**) are hash primitives and included in the signature so they cannot be forged. The integrity of **Sig** is checked against fabrication: neither the network nor the hypervisor have access to the private signing key $\mathbf{SK_P}$ and the customer has access to a chain of certificates that allows for him or her to verify the signature. It is also checked against replay of values: the

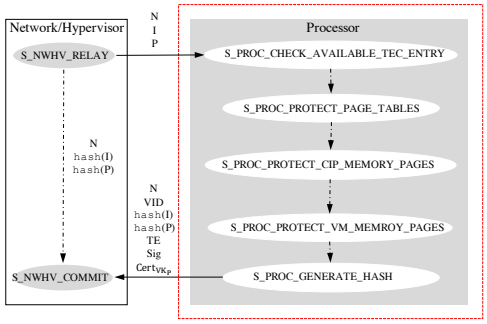

Figure 4: Model of VM Launch Mechanisms. $\mathbf{Cert_{VK_P}}$ is the certificate of the signing key used by the `processor` in creating the signature Sig.

customer can check the nonce, **N**, that he or she generated for this run of the protocol.

3.2 Internal Interaction: VM Launch

We now show how to do the security verification of setting up protections for the VM's memory pages (protection initialization phase in Figure 1).

Modeling. Figure 4 shows the flow chart of the VM Launch mechanism. The mechanism is triggered when the hypervisor tries to start a new VM, as part of the VM startup attestation external protocol. The hypervisor sets up the VM and then executes the `vm_launch` instruction. The processor captures this instruction and atomically launches the VM with the following five operations, highlighted in Figure 4:

(1) The processor consults the TEC tables to find a free entry where the information about the VM will be stored. (2) Once a free VM entry is found, the page tables are protected. (3) Then the Confidentiality and Integrity Protection (CIP) tables for the VM's pages are protected. (4) The VM's pages are protected. Each memory page is protected by denying access to the hypervisor and to DMA. (5) Finally, the hashes of the VM image and VM protections are generated. The page table page count is saved in the TEC table entry for the VM, and the VM is actually launched.

Security invariants. We identify one invariant:

① The processor needs to ensure the VM started has exactly the configuration and protection requested, and that correct hash measurements of the VM are taken.

Preconditions. We require several preconditions about the processor, these are a subset of the preconditions needed by the prior external protocol.

(C1) The processor is trusted.
(C2) The processor has valid $\mathbf{Cert_{VK_P}}$ and $\mathbf{SK_P}$.

Implementation. As above, we model the untrusted network and untrusted hypervisor as a single entity, with the capability to fabricate values and replay values. The processor is trusted based on our preconditions. We use Murphi to model the processor as a state machine. The `Processor` needs to ensure the integrity of the start up values received when a request to launch a VM is received: **N**, **VID**, `hash`(**I**), `hash`(**P**), **TE**, **Sig** and $\mathbf{Cert_{VK_P}}$. We tag these values as requiring integrity protection and check if these values are fabricated or replayed when the protocol reaches the commit state.

Results. This protocol focuses on integrity of the start up values received: **N**, **VID**, `hash`(**I**), `hash`(**P**), **TE**, **Sig** and $\mathbf{Cert_{VK_P}}$. The model keeps track of whether the reads or writes to protection tables were accessed only by the trusted hardware. Our verification results indicate that the processor will correctly conduct the above five steps, and generate the correct hash measurements at the commit state.

3.3 Security Discussion

Coverage. In addition to the two protocols shown above, five other protocols or interactions were verified, as listed in Table 1. The protocols and interactions verified cover the execution phases from Figure 1, except for VM migration. The methodology facilitates a "design for security" approach where architects can validate individual protocols and interactions at the design phase.

Impact. The verification effort uncovered two flaws in the original design [33], and later fixed in [32]. The first was a replay attack in the VM Suspend and Resume protocol. The original design [33] included a nonce to prevent replay attacks. However, when modeling the internal interaction due to VM Suspend & Resume, the verification of the model failed, pointing out that the "nonce" value was not updated during the suspend and resume operation as originally assumed, thus not providing replay protection. A related problem was discovered about the trust evidence data, previously also only stored in registers. Stale trust evidence data could have been sent back to the customer, by a compromised hypervisor.

4 VERIFYING A DISTRIBUTED SYSTEM

CloudMonatt [36] is a flexible distributed cloud architecture to monitor and attest the security health of customers' VMs in the cloud. Figure 5 shows the architecture overview of CloudMonatt. It involves four entities: the customer, the Cloud Controller, the Attestation Server and the cloud server. The Cloud Controller acts as the cloud manager, responsible for taking VM requests and servicing them for each customer. The Attestation Server acts as the attestation requester and appraiser, to collect the security measurements from the VM, interpret the measurements and make attestation decisions. The Cloud Server has a `Monitor Module` which contains different types of monitors to provide comprehensive and rich security measurements. It has a `Trust Module` responsible for server authentication, secure measurement storage and crypto operations.

We now show how our security verification methodology can be used to verify the main attestation protocol of CloudMonatt. We also show how this methodology can help to narrow down the number of trusted components needed in the trusted computing base for this distributed system.

4.1 External Protocol: Cloud Attestation

Cloud attestation is the *procedure of making unforgeable claims about the security conditions of customers' VMs based on the evidence supplied by the host server.* We verify that the requested report is not tampered with in CloudMonatt architecture, and hence the integrity

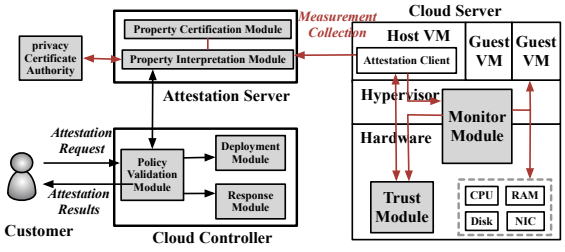

Figure 5: Architecture of CloudMonatt.

of the end-to-end attestation is achieved (protection attestation phase in Figure 1).

Modeling. We model each entity involved in this distributed system as an interacting state machine, as shown in Figure 6. The whole process starts from the customer, who sends to the Cloud Controller the attestation request including the VM identifier, **VID**, and the security properties, **P**. Then the Cloud Controller forwards the request to the Attestation Server, with the host servers identifier, **I**. The Attestation Server sends **MR**, the request of necessary measurement, to the host server. The cloud server collects the required measurements **M**, hashes and signs the measurements, and sends them back to the Attestation Server. The Attestation Server checks the received message and, if correct, generates the attestation report **R** based on **M** and **P**. Then the Attestation Server signs the report and transmits it to the Cloud Controller. The Cloud Controller checks the message and, if correct, hashes and signs the report, and sends it to the customer. The customer ends the attestation session if the he finds the report is correct.

Security invariants. We identify one invariant:

① The attestation report **R** the customer receives is indeed the one for **VID** with **P**, specified by the customer.

Preconditions. Initially, we specify several preconditions and check if the above invariant can be satisfied under these preconditions. Later, we verify each of these preconditions.

(C1) The cloud server is trusted.
(C2) The Attestation Server is trusted.
(C3) The Cloud Controller is trusted.

Implementation. We model the external protocol in ProVerif. Specifically, we declare each subject as a process. Each process keeps some variables. If the subject is trusted, we denote these variables as `private`, not accessible by the attacker. Otherwise the variables are assumed public. We declare a `network` connected between each pair of subjects, to represent the untrusted communication channels. These channels are under full control of the network-level adversaries, who can eavesdrop or modify any messages. We use the cryptographic primitives from ProVerif to model the public key infrastructure for digital certificate, authentication and key exchange. Then we model the attestation process for an unbounded number of sessions, and check if the adversary can compromise the integrity of the report in any session.

We use ProVerif's *reachability proof* functionality to verify the integrity of a message. Specifically, we define a function $\mathbb{R}$(**VID**, **P**) to denote the correct report of VM **VID** for property **P**. At the customer's state "S_CUST_COMMIT", the customer receives the report **R**, and we check if the statement **R** = $\mathbb{R}$(**VID**, **P**) is always true. We use the statement "`query event`(**R**≠$\mathbb{R}$(**VID**, **P**))" to check the negative scenario: an integrity breach has occurred. If this query statement is false, the attacker has no means to change the message **R** without being observed by the customer and the integrity of **R** holds.

Results. First, ProVerif shows the security invariant ① is satisfied under the preconditions (C1) – (C3). The network-level adversaries cannot compromise the integrity of the messages without being observed, as all the messages are cryptographically protected. Second, ProVerif shows that preconditions (C1) – (C3) are necessary to keep the invariants correct, and missing any precondition can lead to violations of the invariant. An untrusted cloud server can counterfeit wrong measurements, making the customer receives wrong attestation report generated from the measurements. An untrusted Attestation Server can generate wrong attestation report for the customer. An untrusted Cloud Controller can modify the attestation reports before sending to the customer.

4.2 Internal Interaction: Evidence Collection

Placing the entire server into the TCB would require stronger security protection, which is expensive and difficult to achieve. So, we conduct *internal verification* to identify the necessary components inside the servers that need to be trusted. We verify the evidence collection process in the cloud server (protection attestation phase in Figure 1).

Modeling. We model the key components inside a cloud server as state machines (Figure 7). We also include the untrusted network as the initiator and finisher subject in the internal protocol to interact with the server. The whole process starts when the network passed the encrypted measurement request to the server. The `Attestation Client` processes the request and passes it to the `Monitor Module`. The `Monitor Module` collects the correct measurements, and then stores the measurements together with other related information in the `Trust Module`. The `Trust Module` calculates the hash and signature using its private attestation key. Then the signature is encrypted by the `Attestation Client` and sent out. The network goes to the commit state when it receives the encrypted measurement.

Security invariants. We identify one invariant:

① The cloud server needs to ensure that the correct measurement **M** are taken for VM **VID** with request **MR**.

Preconditions. We identify a set of possible preconditions.

(C1) The `Monitor Module` is trusted.
(C2) The `Trust Module` is trusted.
(C3) The channel between the `Monitor Module` and the `Trust Module` is trusted.

Implementation. We model a software or hardware component as a process. Each component keeps some variables and operates as a state machine. If one component is in the TCB, then its variables will be declared as `private`. Otherwise its variables are public to attackers. If two modules are linked by an untrusted channel, then we declare a public `network` between these two components.

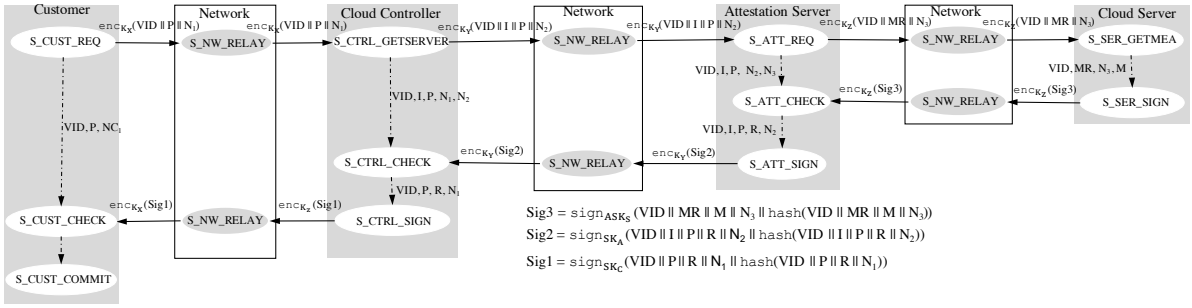

Figure 6: The external protocol in *CloudMonatt*. SK_C, SK_A and ASK_S are the private signing keys of the Cloud Controller, the Attestation Server and the cloud server, respectively. K_X, K_Y and K_Z are symmetric keys between the customer and the Cloud Controller, between the Cloud Controller and the Attestation Server, and between the Attestation Server and the cloud server, respectively.

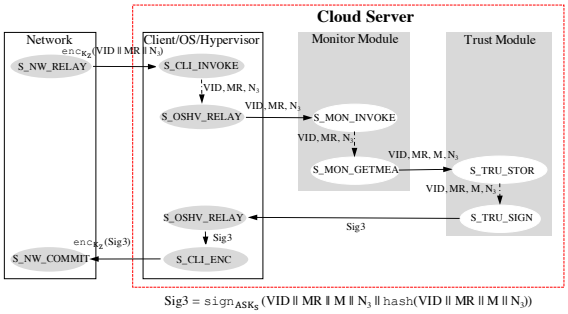

Figure 7: Internal interactions in the cloud server. K_Z is the symmetric key known to the Attestation Server and the cloud server. ASK_S is the private signing key of the cloud server.

Otherwise we combine the two component into one process so that they can exchange messages securely.

We also use ProVerif's reachability proof functionality to verify the integrity of measurement **M**. When the network reaches state "S_NW_COMMIT", we denote the measurement inside the encrypted message as **M**. We also defines a function $\mathbb{M}$(**VID**, **MR**), which gives the correct measurement of VM **VID** for the measurement request **MR**. Then we check if the statement **M** = $\mathbb{M}$(**VID**, **MR**) is always true at the commit state. We use the statement "`query event(`**M**$\neq\mathbb{M}$(**VID**, **MR**)`)`" to discover potential integrity breach. If this statement is false, the attacker has no means to change **M** without being observed by the customer and the integrity of **M** holds.

Results. We verify that it is sufficient and necessary to keep the security invariant with these preconditions, when the network, OS and hypervisor is untrusted. Missing any prediction can lead to invariant violation: an untrusted `Monitor Module` can collect wrong measurements **M** and store them into the `Trust Module`; an untrusted `Trust Module` can generate a fake signature over any measurements using the signing key ASK_S; an untrusted channel between the `Monitor Module` and `Trust Module` gives the adversary a chance to modify the measurements without being detected.

4.3 Security Discussion

Coverage. We show the main CloudMonatt attestation protocol is secure, i.e., correct and unforgeable. We show the evidence collection process in the cloud server is secure. We also verified the property interpretation process in the Attestation Server and the health checking process in the Cloud Controller in the same way as we showed for the Cloud Server. This completes the end-to-end security verification of the protection attestation phase in CloudMonatt.

Model	Int. or Ext.	Lines of Code	Runtime (s)
VM Startup	Ext.	1159	0.8s
VM Launch	Int.	462	0.6s
VM Secure Channel	Ext.	1332	0.3s
VM Trust Evidence	Ext.	1081	0.2s
VM Suspend & Resume	Ext.	1054	0.5s
VM Mem. Update	Int.	687	0.7s
VM Terminate	Int.	417	0.8s

Table 1: HyperWall verification evaluation results.

Impact. One of the most interesting results of security verification is to show how we can enhance the security of the architecture during design. In CloudMonatt, it showed that only the Monitor Module and Trust Module of a cloud server should be included in the TCB. Normally, third party customers (at guest VM privilege) has no capability to subvert the security functions provided by these two modules (at the hypervisor privilege). To defeat attacks (e.g., privilege escalation) caused by the vulnerabilities of the original system, secure enclaves can be used to protect the execution environment of the Monitor Module and Trust Module, leveraging mechanisms provided by Bastion [7].

5 VERIFICATION EVALUATION

In addition to the protocols presented in this paper, we have also verify five more for HyperWall and two more for CloudMonatt. For HyperWall, we use CMurphi 5.4.4 and the models were run with options `-tv -ndl -m1000`. The `-tv` writes a violating trace (if an invariant fails), and the `-ndl` disables the checking for deadlock states. For CloudMonatt we use ProVerif 1.88 with default options.

The collected results for HyperWall in Table 1 and for CloudMonatt in Table 2. The verification process is iterative, where the ProVerif or Murphi files may be updated many times, thus comments are crucial to understand the development of the verification strategy. We can also observe that the verification runtime is also very small: due to the breakdown of internal and external verification, we can verify complex architectures within a very short time. The most effort-consuming step is the design and writing of the verification models, but the actual verification is quick.

Model	Int. or Ext.	Lines of Code	Runtime (s)
External	Ext.	262	0.2s
Evidence Collection	Int.	123	0.1s
Property Interpretation	Int.	205	0.2s
Health Checking	Int.	187	0.1s

Table 2: CloudMonatt verification evaluation results.

6 RELATED WORK

Secure application verification. Past work, e.g., [26, 29], has focused on verifying software with respect to an ISA. In contrast, we are going one layer below, focusing on the state machines and protocols of the hardware.

Secure architecture verification. Of the different secure architectures, XOM [20] and SecVisor [25] have received the benefit of security verification using model-checking. In industry, e.g., the IBM 4758 cryptographic co-processor's design included security verification [27]. These works, however, have not focused on external and internal protocols and interactions as we do.

Security verification tools. A number of speciality tools exist for security verification and verification of security protocols. These tools include HERMES [6], Casrul [9], AVISPA [3], Scyther [11], ProVerif [4], etc. Various model checkers have also been used in security verification. Typical model checkers include Maude [15], Alloy [16], Murphi [14], CSP [22], FDR [23], etc. Our work does not invent a new tool, rather it shows how architects can leverage existing tools. We show that both the protocol verification tools (ProVerif) and model checkers (Murphi) can be used by our framework to implement the security verification task. We enhanced Murphi with automatic checking for integrity and confidentiality, but did not need to make any changes to Proverif.

Security verification methodologies. As an alternative to modelling, projects such as Caisson [18] or SecVerilog [35] work directly with the hardware source code, and leverage information flow tracking to analyze potential information leaks in an architecture. Our work does not require hardware source code, and can be complimentary to the approaches that work with HDL code.

7 CONCLUSION

We present a security verification methodology, which is applicable to different security architectures and systems. We break the verification task into external verification and internal verification to achieve scalability of verification. For each type of verification, we propose the methodology for modeling the system and the attackers, deriving security invariants, and creating the implementation. We use two case studies to evaluate our methodology: security verification of a standalone processor architecture, HyperWall, and verification of a distributed cloud system, CloudMonatt. Our case studies show that we can verify the design of complex secure architectures efficiently, discover and fix bugs, and enhance the security of the design. We hope that our methodology can be easily adopted by computer architects to verify the security of their designs, and to do more research in the important area of security verification methodologies.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments. This work was supported in part by NTU Startup Grant, and NSF grants 1813797 and 1651945.

REFERENCES

[1] [n.d.]. Event-B and the Rodin Platform. http://www.event-b.org/.
[2] AMD. [n.d.]. AMD Memory Encryption. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, accessed May 2016.
[3] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. 2005. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*. 281–285.
[4] Bruno Blanchet. 2005. ProVerif automatic cryptographic protocol verifier user manual. *CNRS, Departement dInformatique, Ecole Normale Superieure, Paris* (2005).
[5] Bruno Blanchet. 2012. Security Protocol Verification: Symbolic and Computational Models. In *International Conference on Principles of Security and Trust*.
[6] Liana Bozga, Yassine Lakhnech, and Michaël Périn. 2003. HERMES: An Automatic Tool for Verification of Secrecy in Security Protocols. In *Computer Aided Verification*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Lecture Notes in Computer Science, Vol. 2725. Springer Berlin Heidelberg, 219–222. https://doi.org/10.1007/978-3-540-45069-6_23
[7] David Champagne and Ruby B. Lee. 2010. Scalable architectural support for trusted software. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*. 1–12. https://doi.org/10.1109/HPCA.2010.5416657
[8] Yu-Yuan Chen, Pramod A Jamkhedkar, and Ruby B Lee. 2012. A software-hardware architecture for self-protecting data. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 14–27.
[9] Véronique Cortier and Bogdan Warinschi. 2005. Computationally Sound, Automated Proofs for Security Protocols. In *Programming Languages and Systems*, Mooly Sagiv (Ed.). Lecture Notes in Computer Science, Vol. 3444. Springer Berlin Heidelberg, 157–171. https://doi.org/10.1007/978-3-540-31987-0_12
[10] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 857–874.
[11] Cas J.F. Cremers. 2008. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Lecture Notes in Computer Science, Vol. 5123. Springer Berlin Heidelberg, 414–418. https://doi.org/10.1007/978-3-540-70545-1_38
[12] Cas J.F. Cremers. 2008. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security* (Alexandria, Virginia, USA). ACM, New York, NY, USA, 119–128. https://doi.org/10.1145/1455770.1455787
[13] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. 2007. Protocol Composition Logic (PCL). *Electronic Notes in Theoretical Computer Science* 172, 0 (2007), 311–358. https://doi.org/10.1016/j.entcs.2007.02.012
[14] David L. Dill. 1996. The Murphi Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*. 390–393.
[15] Steven Eker, JosŽ Meseguer, and Ambarish Sridharanarayanan. 2004. The Maude LTL Model Checker. *Electronic Notes in Theoretical Computer Science* 71, 0 (2004), 162–187. https://doi.org/10.1016/S1571-0661(05)82534-4
[16] Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.
[17] Ruby B. Lee, Peter Kwan, John Patrick McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 2–13.
[18] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: a hardware description language for secure information flow. *ACM Sigplan Notices* 46, 6 (2011), 109–120.
[19] David Lie, John C. Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. 2003. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of Symposium on Security and Privacy (S&P)*. 166 – 177.
[20] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *SIGPLAN Not.* 35 (November 2000), 168–177. Issue 11. https://doi.org/10.1145/356989.357005
[21] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *ACM Intl. Workshop on Hardware and*

Architectural Support for Security and Privacy.

[22] Andrew W. Roscoe, C. A. R. Hoare, and Richard Bird. 1997. *The Theory and Practice of Concurrency*. Prentice Hall.

[23] Andrew W. Roscoe and Zhenzhong Wu. 2006. Verifying Statemate Statecharts Using CSP and FDR. In *Formal Methods and Software Engineering*, Zhiming Liu and Jifeng He (Eds.). Lecture Notes in Computer Science, Vol. 4260. Springer Berlin Heidelberg, 324–341. https://doi.org/10.1007/11901433_18

[24] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. 2012. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, 78–94.

[25] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Oper. Syst. Rev.* 41, 6 (2007), 335–350. https://doi.org/10.1145/1323293.1294294

[26] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2015. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1169–1184.

[27] Sean Smith, Ron Perez, Steve Weingart, and Vernon Austel. 1999. Validating a High-Performance, Programmable Secure Coprocessor. In *Proceedings of the 22nd National Information Systems Security Conference (NISSC)*.

[28] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17.

[29] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. 2017. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2435–2450.

[30] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. 160–171. https://doi.org/10.1145/782814.782838

[31] Jun Sun, Yang Liu, and JinSong Dong. 2021. PAT: Process Analysis Toolkit. https://pat.comp.nus.edu.sg/.

[32] Jakub Szefer. 2013. *Architectures for Secure Cloud Computing Servers*. Ph.D. Dissertation. Princeton University.

[33] Jakub Szefer and Ruby B. Lee. 2012. Architectural Support for Hypervisor-Secure Virtualization. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 437–450.

[34] Johannes Winter. 2008. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*. 21–30.

[35] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A hardware design language for timing-sensitive information-flow security. *Acm Sigplan Notices* 50, 4 (2015), 503–516.

[36] Tianwei Zhang and Ruby B. Lee. 2015. CloudMonatt: An Architecture for Security Health Monitoring and Attestation of Virtual Machines in Cloud Computing. In *ACM International Symposium on Computer Architecture*.

[37] Tianwei Zhang and Ruby B. Lee. 2016. Monitoring and Attestation of Virtual Machine Security Health in Cloud Computing. *IEEE Micro* 36, 5 (2016).

[38] Tianwei Zhang and Ruby B Lee. 2017. Design, implementation and verification of cloud architecture for monitoring a virtual machine's security health. *IEEE Trans. Comput.* 67, 6 (2017), 799–815.

[39] Tianwei Zhang, Jakub Szefer, and Ruby B. Lee. 2012. Security Verification of Hardware-enabled Attestation Protocols. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.