# Security Verification of Secure Processor Architectures and Systems

Jakub Szefer
Yale University
jakub.szefer@yale.edu

Tianwei Zhang
Princeton University
tianweiz@princeton.edu

Ruby B. Lee
Princeton University
rblee@princeton.edu

## ABSTRACT

In the last decade, a number of secure processor architectures have been proposed in academia, and now some are available in consumer products, such as Intel's SGX or AMD's SEV. However, most, if not all, of the designs are not thoroughly security verified, bringing into question the security of these architectures, and systems built around them. To address this issue, in this paper we present a new and practical framework for security verification. Specifically, we break the verification task into external verification and internal verification. External verification considers the external protocols, i.e. interactions between management servers, compute servers, trusted third parties, and the customers. 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, a distributed system or a large-scale cloud system. We evaluate our verification method on the CloudMonatt [1] and HyperWall [2] architectures, using both ProVerif and Murphi verification tools, as examples.

## 1. INTRODUCTION

Over the last decade, a number of secure processor architectures have been designed such as XOM [3], AEGIS [4], SP [5, 6], Bastion [7], or HyperWall [2]. They all share a common feature that processor hardware is enhanced and entrusted with implementing some security functionalities, such as isolation of trusted software modules from untrusted operating systems, or protecting virtual machines from untrusted hypervisors, for example. Ideas presented by these architectures have been recently implemented in commercial designs, such as Intel's SGX [8, 9] or AMD's SEV [10].

Once any such security architecture is designed, it needs to be checked to make sure there are no security vulnerabilities with the design that could allow an attacker to subvert the protections. Especially, once hardware is manufactured, it is expensive and often simply not possible to update or replace it. Unlike software-based solutions which may be easily patched in the field, hardware-based protections need to be correct from the beginning since only limited hardware or firmware workarounds are possible in the field. 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.

Our contribution in this paper is the definition of one such general-purpose security verification framework. The presented approach breaks down a system into smaller components for verification and is the first to propose that for security architectures this breakdown can be achieved effectively by focusing on external verification, of the external protocols, and internal verification, of the internal interactions. External protocols are used for communication with the outside world, while internal interactions are for communication among components within the system itself. The components in these protocols and interactions are made up of various software or hardware mechanisms that themselves need to be verified. State machines and logic of these components can be verified by standard functional verification methodologies.

A complex secure processor today is likely to be used in a cloud computing scenario where there is a small number of centralized management servers, a large number of computing servers, and finally the end users. All of these are connected by, possibly untrusted networks, and require mutual communication to achieve secure operation. The external protocols represent the interaction of the different parties, across untrusted networks and while being relayed by untrusted parties. Secure architecture verification cannot thus only focus on the secure processor itself. Verification of complex architectures needs to be achieved by focusing on two additional key aspects they all share: external protocols and internal interactions.

Our methodology is realized in both ProVerif [11] and Murphi [12], showing that it is flexible and can be done in tools already familiar to designers. ProVerif is a well-known protocol verifier, which can handle numerous in-

Jakub Szefer, Tianwei Zhang, Ruby B. Lee, "Security Verification of Secure Processor Architectures and Systems," Princeton University Technical Report CE-L2017-002, August 2017.

teracting principals and has built-in support for cryptographic operations; meanwhile, Murphi is a generic model checker that can model interacting principals as state machines. They both, independently, can be used to do the verification following our methodology, giving designers flexibility to choose the tool they prefer. This methodology has been partly used to help design and enhance the CloudMonatt [1] and HyperWall [13] architectures, further showing its practicality and granularity. 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.
- 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.

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 discuss the evaluation results in Section 6. We summarize related work in Section 7 and conclude in Section 8.

# 2. VERIFICATION APPROACH

*The security verification of security architectures goes beyond functional verification.* We need to worry about potential attackers who will try to intelligently manipulate inputs, outputs or any state they can access, to subvert the protections. During the 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 be able to capture these untrusted components (and their capabilities), and to model the interaction among these trusted and untrusted components.

A security 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. To achieve the scalability of verification, it is necessary that the verification is done on each part of the system, rather than on the whole architecture at once. Still, the verification of the sub-parts must compose into the verification of the whole system.

In summary, necessary features of the security verification methodology are the ability to:

- Model both trusted and untrusted components and their interactions.
- Specify the potential attack capabilities of untrusted components, and have the methodology check automatically for the possible attacks.
- Specify security invariants for different security properties.
- Verify parts of an architecture, rather than the entire system at once, to avoid the state explosion problem.
- Ensure that the verification of the parts can eventually compose to the security verification of the whole security architecture.

We propose a method of breaking the security verification of a system into smaller verification tasks, i.e., external verification and internal verification, in Section 2.1. Then we describe the detailed steps to conduct each verification task in Section 2.2. One significant feature of our method is that we can re-use the existing tools often used for functional verification, and apply them to the security verification task. In particular, the external protocols and internal interactions can be modelled as sets of interacting principals. And their communication can be verified using ideas from protocol verification or model checking.

Note that the functionality of the mechanisms which implement the protections can be modelled and checked already with the existing functional verification tools. What we do propose and demonstrate is that these techniques and tools used for functional verification can be applied to the security verification task. Functional verification is thus not considered in this work.

## 2.1 External and Internal Verification

A system is composed of many components, including external and internal components of a server. Each component is realized by one or more mechanisms. These mechanisms can be hardware or software. We specify *external protocols* as the interaction of the system including 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. Interestingly, 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 (see notes in Section 2.1.2).

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. The components, even whole servers, can be treated as a blackbox during external verification – and in turn checked during internal verification steps.

### 2.1.1 Identifying Protocols and Interactions

We focus on modeling and verifying external protocols and internal interactions. To find the different security-sensitive interactions, we need to identify different execution phases of a secure architecture or sys-

Table 1: The different execution phases of a hardware secure processor architecture.

| Phase | Description |
|---|---|
| 1) System Startup | When any protection mechanisms are setup, software and hardware state is reset (for example the memory holding protections metadata is cleared) and any bootup-time measurements of the trusted computing base (TCB) are taken. This is independent of any setup and runtime of protected software (code or data) that will run later. |
| 2) Protection Initialization | When the protections for the code or data are initialized. This depends on some inputs related to the protected code or data (e.g., customer's requested protections) as well as the code and data itself. This is also when any code or data start up measurements are taken (e.g., hashes for later attestation). |
| 3) Code and Data Runtime | At runtime, when the protections are actively enforced and any runtime attestation measurements are taken. |
| 4) Configuration Update | At runtime, when any change to the protections or to the configuration (e.g., memory map for the code is updated) triggers updates to the protection state as well as potentially triggers some measurements. |
| 5) Code and Data Migration | At runtime, when code and data are migrated from one system to another, potentially triggering protection initializaiton steps on the destination host, and code and data termination steps on the source host. |
| 6) Code and Data Termination | When the code and data are terminated. Any cleanup and scrubbing of sensitive code or data (and any metadata) takes place at this point in time. |
| 7) System Power Down | When any final cleanup takes place before the system shuts down. |

tem. For such architectures, we propose that there are seven main phases into which the execution can be broken down to, listed in Table 1. The middle five 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.

#### 2.1.2 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 composition in our work. We focus on providing a sound methodology for verification of individual external protocols and internal interactions.

However, recently Protocol Composition Logic (PCL) has been proposed [14]. The composition theorems in PCL allow proofs of complex protocols to be built up from proofs of their constituent sub-protocols. PCL has actually been realized using Murphi, which is one of the tools that can be used to realize our methodology. 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 Verification Framework

To verify a system's protocols and operations, we first specify the verification goals and invariants based on the system's functionality. Then we build models for the system, and identify the trusted and untrusted subjects in the system. We implement the models and verification invariants in a protocol verifier or a model checker and run the tools to test if the invariants pass for every possible path 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 Essential System Components

For each protocol or interaction, a designer has to enumerate the components or principals involved:

- Hardware components – these are the hardware components of the system, often microprocessor, memory chips, co-processors, etc.
- Software components – these are the software components of a system, often the hypervisor, operating system(s) and applications.
- Network components – there are the means by which physically separate components communicate.
- Internal communication components – communication within a system is achieved via hardware components such as memory buses or instructions.
- Customer principal – there is often a customer or user involved in the operation of the system, e.g., remote user who requested a VM.
- Trusted third party principal – there may be a trusted third party, such as an attestation service that handles some sensitive data.
- Cloud provider principal – there will typically be the cloud provider that may be untrusted and who guards access to the remote servers, in some cases the cloud provider can be subsumed in the (untrusted) network component.

The network component, customer, and (if needed) trusted third party and cloud provider are explicitly included for the benefit of the external protocols which involve the remote customer connecting via communication path to the server. The network is one component in our models, but our verification could be extended to look at the security of its sub-components (routers, switches, etc.).

These essential components are the principals in the external protocols and internal interactions. Each principal's operation can be represented as states of a state machine, and communication among the principals can

be represented as messages sent between the principals.

### 2.2.2 Symbolic Modeling

For the verification, we adopted the symbolic modeling method [15], where the cryptographic primitives are represented by function symbols and perfect cryptography is assumed. Specifically, we first specify subjects involved in this verification procedure. A subject can be a customer or a server in the distributed system, or a hardware or a software module inside a server. For the *external verification* of protocols, since we treat each server as a blackbox, we model each server and the customer as a subject. For *internal verification* of the interactions within a server, we need to consider the internal activities inside the server, so we model each software and hardware module involved in the system operation 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 and a *finisher subject* that ends the protocol, 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, and no security invariants are violated at any time.*

### 2.2.3 Confidentiality Validation

During the execution of the model, each principal has access to various values, including ones tagged as confidential to indicate the need for confidentiality protection of that value. When the final state of an untrusted principal has been reached, the untrusted principal has seen all the inputs. Now, 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 encrypted cipher text in some state, and decryption key in another).

For each value tagged as confidential, the invariants check that the value is also tagged as encrypted (i.e. it has a decryption key associated with it). If a value is not encrypted, a blatant confidentiality violation exists, if this plaintext value reaches an untrusted principal. For each value tagged as confidential and encrypted, the invariants search through all the other values to see if there is a decryption key . If the untrusted principal has access to an encrypted value and the key, it can obtain the plaintext, thus violating confidentiality. 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.

These invariants are evaluated when the last state of each untrusted principal is reached. The invariants could also be evaluated earlier and the model could stop as soon as the first invariant fails. Given, however, that our models run quite quickly, on the order of 1 second, evaluation at the ending "commit" state of an untrusted principal is reasonable and easier to implement.

### 2.2.4 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. Thus the model has visibility into the sources of the inputs and how these sources could have modified them. During a run of the model, it can be checked 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:

- 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 (labeled `private`),
- the checked values should include a nonce value for freshness, and
- checking for self-consistency of values, which allows a trusted party to check the inputs and make sure they are mutually consistent.

### 2.2.5 Cryptographic Protocol Validation

The external protocols, as well as internal interactions, use cryptographic primitives such as encryption, hashing, digital signatures, and public-key cryptography. Although we adopted the symbolic modeling method, where the cryptographic primitives are represented by function symbols and perfect cryptography is assumed, still, the protocol needs to be verified. Cryptographic protocol verification is done by checking messages in the protocols with "known-good" values, also noted as `private` in the models. These values, such as certificates or private keys, are known to be correct (through the assumed preconditions) and not accessible to the attacker. With these known-good values, a principal in the protocol can verify received messages. Freshness of the protocols is achieved through use of nonces. If a principal has no access to a know-good nonce, then it may accept a stale or replayed message. In effect, the preconditions can be reduced to known-good `private` values that are necessary and sufficient to validate each run of the protocol.

### 2.2.6 Five Verification Steps

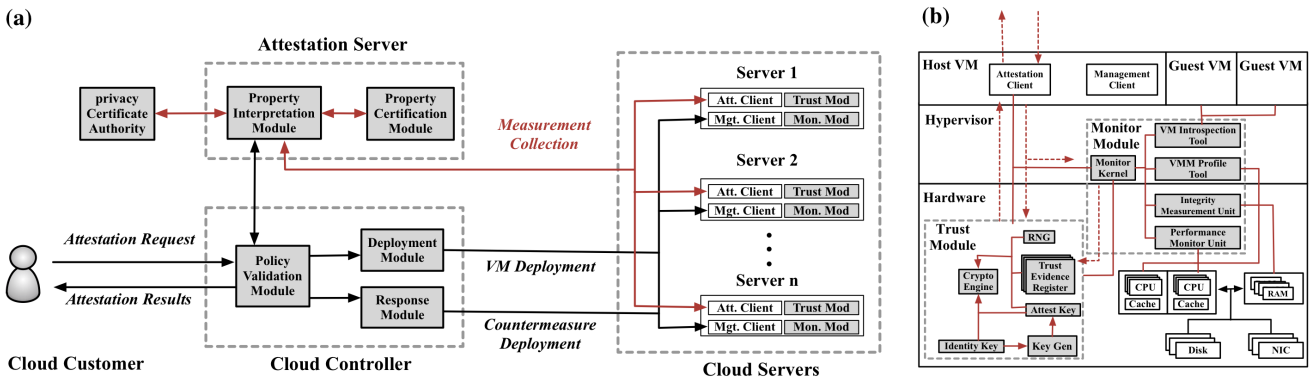


Figure 1: Architecture of CloudMonatt, shown in (a), and details of the Monitor and Trust Modules in a cloud server, shown in (b). Grey blocks are the trusted.

**Modeling.** The external protocol or internal interaction is first described by a set of components or principals. Each component or principal is represented by a set of states. The components' or principals' states are connected by edges to capture passing of values between the states. 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.

**Security invariants.** Each protocol or interaction needs to satisfy certain security invariants – it is only verified if for all possible execution traces, the invariant is not found to be violated. 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.

**Preconditions.** The protocols and interactions are subject to certain constraints, which we call the 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 (some preconditions initially thought to be required are found to be superfluous as verification passes with these preconditions removed).

**Implementation.** We show that these three steps can be realized in both ProVerif [11] and Murphi [12] exemplifying that this is a flexible methodology. The principals are implemented as sets of interacting state machines, with invariants realized using built-in features of ProVerif or Murphi. Preconditions are the assumptions used when creating the verification models.

**Results** . The verification results are either 1) 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 assumed to be finished.

In the following two sections, we present examples of our verification methodology. We validate it on two types of secure architectures: a distributed cloud system (CloudMonatt [1, 16]) and a secure processor (HyperWall [2, 13]).

## 3. VERIFICATION OF CLOUDMONATT

CloudMonatt [1] is a flexible distributed cloud architecture to monitor and attest the security health of customers' VMs in the cloud. It involves four entities: the customer, the Cloud Controller, the Attestation Server and the cloud server. Secure communication protocols and secure key management are used between different entities to ensure the monitoring activities happen in an unforgeable way. Each entity has a public-private key pair, and uses the private key for signing. Figure 1 shows (a) the architecture overview of CloudMonatt, and (b) the details of its 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. When a customer requests to monitor the security properties associated with his or her VM, the Cloud Controller entrusts the Attestation Server to collect the security measurements from the VM, interpret the measurements and make attestation decisions. Then the Attestation Server sends a report back to the Cloud Controller. The Cloud Controller then sends the results back to the customers to keep them informed of their VMs' security health. Difficulties include the fact that the cloud customer does not (and should not) know which cloud server his VM is running; there are many security properties that can be monitored, and there is a semantic gap between the security properties the customer is interested in and the measurements that can be taken from servers.

The Cloud Server includes an `Attestation Client` for taking requests from the Attestation Server. It has a `Monitor Module` which contains different types of mon-

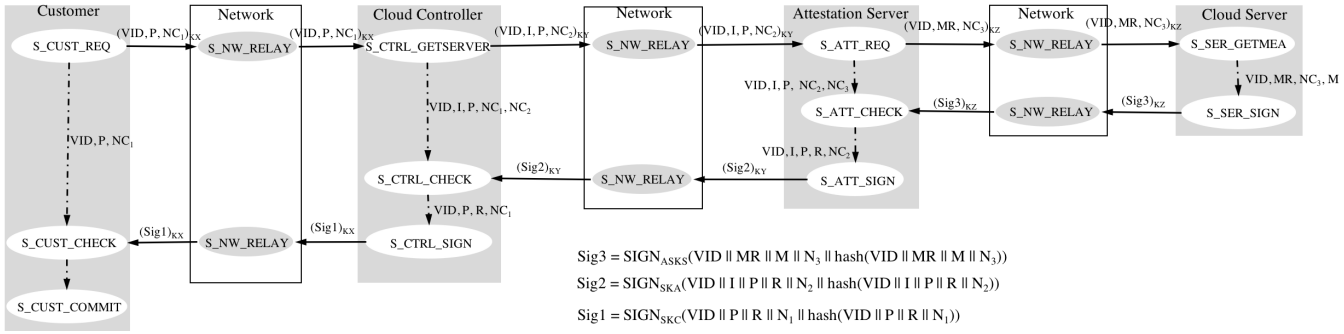


Figure 2: The external protocol in *CloudMonatt*. The gray blocks are trusted parties while the white blocks are untrusted parties. **KX**, **KY** and **KZ** are symmetric keys between the customer and the Cloud Controller, between the Cloud Controller and the Attestation Server, and between the Attestation Server and the cloud server. **SKC**, **SKA** and **ASKS** are the private signing keys of the Cloud Controller, the Attestation Server and the cloud server. $\mathbf{NC}_1$, $\mathbf{NC}_2$ and $\mathbf{NC}_3$ are the nonces used by the customer, the Cloud Controller and the Attestation Server.

itors to provide comprehensive and rich security measurements. It has a `Trust Module` responsible for server authentication, secure measurement storage and crypto operations (e.g., encryption, signatures generation).

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 in this distributed system.

## 3.1 External Protocol: Cloud Attestation

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 are only interested in the attestation protocol. We do not consider other activities within the cloud system.

**Modeling.** We model each entity involved in this distributed system as an interacting state machine, as shown in Figure 2. Each subject is made up of some states. The customer is the initiator as well as the finisher subject. The whole process starts from the customer side, who sends to the Cloud Controller the attestation request including the VM identifier, **VID**, and the desired security properties, **P**. Then the Cloud Controller discovers the host cloud server, and forwards the request to the Attestation Server, with the server identifier, **I**. The Attestation Server identifies the necessary measurements and sends the measurement request **MR** to the host cloud server. The cloud server collects the required measurements, calculates the hash value of the measurements requested, signs the measurements and sends these values back to the Attestation Server. The Attestation Server checks the signature, the hash value and the nonce: if this check is correct, it interprets the measurements and the property, and generates the attestation report, **R**. Then the Attestation Server signs the report and transmits it to the Cloud Controller. After receiving the report, the Cloud Controller checks the signature, the hash value and the nonce. If the check is correct, the Cloud Controller hashes and signs the report, and sends the report to the customer. If the customer finds the encrypted signature of the report is correct, it ends the attestation request session.

**Security invariants.** We identify one invariant:

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

**Preconditions.** We require several preconditions about each server and check if the above security invariant can be satisfied under these preconditions.

(C1) The cloud server is trusted. It will collect and sign correct measurements;
(C2) The Attestation Server is trusted. It will process the received measurements and generate the reports correctly;
(C3) The Cloud Controller is trusted. It will process the VM health reports correctly.

However, the network in our threat model is not trusted. The adversary can eavesdrop as well as falsify any of the messages transmitted between different servers.

**Implementation.** We model the *external* authentication and communication protocol in ProVerif. Specifically, we declare each subject (the customer, the Cloud Controller, the Attestation Server and the cloud server) as a process. Inside the process we model the operations of state machines shown in Figure 2. Each process keeps some secrets (e.g., cryptographic keys, attestation reports or measurements). If the subject is trusted, then the attacker cannot get these secrets, and we use the keyword `private` to denote these variables. Otherwise the variables are assumed public.

To model the network activities in this distributed system, 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 transmitted in the channels.

We also use the cryptographic primitives from ProVerif to model the public key infrastructure for digital certificate, authentication and key exchange. Then we model all the steps in Figure 2 for an unbounded number of attestation sessions, i.e., the customer keeps sending attestation requests to the cloud system and receiving the reports. ProVerif can check if the adversary can com-

promise the integrity of the report in any attestation session, and display the attack execution trace if a vulnerability is found.

We use ProVerif's reachability proof functionality to verify the integrity property of a message. ProVerif allows us to define an event **E** inside a process at one state, which specifies some conditions. Then we can check if this event will happen when the protocol proceeds using the query statement: "`query event`(**E**)". ProVerif can enumerate all the possible execution traces and check if this event is reachable in some cases. To verify the integrity of the attestation report in invariant ①, we check if the report received by the customer, **R'** is the correct one, **R** when the customer reaches the commit state. Then we establish an event: "(**R'**≠**R**)" at the commit state to denote the integrity breach. We use the statement "`query event`(**R'**≠**R**)" to verify the integrity. If this 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). Even though the network-level adversaries can take control of all the network channels between each server, they cannot compromise the integrity of the messages without being observed, since all the messages are hashed, signed and encrypted before being sent to the network.

Second, ProVerif shows that preconditions (C1) – (C3) are necessary to keep the invariants correct. Missing any precondition can lead to violations of some invariants: if the cloud server is not trusted, then the server-level adversary can counterfeit wrong measurements, causing the Attestation Server to make wrong attestation decisions, and pass them to the Cloud Controller and the customer. If the Attestation Server is not trusted, then it can generate wrong attestation reports for the customer and the Cloud Controller. If the Cloud Controller is untrusted, it can modify the attestation reports before sending to the customer. In all three cases, invariant ① is not satisfied.

Including all the components of the server into the TCB would require stronger security protection for the entire server, which is expensive and difficult to achieve. As such, we conduct the *internal verification* to identify which components inside the servers need to be trusted, to satisfy the preconditions we make in the *external verification*. We show the most complicated case: internal verification of the cloud server. The verification of the Cloud Controller and Attestation Server can be done in a similar way (results in Section 5).

## 3.2 Internal Interaction: Cloud Server

**Modeling.** We abstract the key components inside a cloud server, and model them as state machines, as shown in Figure 3. We also include the Attestation Server to interact with the cloud server. The Attestation Server is the initiator and finisher subject in the internal protocol. The whole process starts when the Attestation Server sends the measurement request to the cloud server. The `Attestation Client` processes the request and passes it to the `Monitor Module`. The `Monitor Module` figures out the corresponding monitor tool and invokes it to collect the correct measurements. Then it stores the measurements together with other related information in the `Trust Module`. The `Trust Module` retrieves the measurements, calculates hash and signature using its private attestation key, which is generated for each new attestation request. Then the signature is encrypted by the `Attestation Client` and sent to the Attestation Server. The Attestation Server will check the hash and signature. It goes to the commit state if the check succeeds.

**Security invariants.** We identify one invariant:

① The measurements **M** the Attestation Server receives are indeed the one for VM **VID** with request **MR**, which were sent to the cloud server.

**Preconditions.** We identify a set of possible preconditions to satisfy the above invariant.

(C1) The `Attestation Client` is trusted, i.e., it can maintain the integrity of the attestation requests and measurements.
(C2) The `Monitor Module` is trusted, i.e., it can invoke the correct measurement and maintain the integrity of the measurements.
(C3) The `Trust Module` is trusted, i.e., it can correctly calculate the hash quote and sign the measurements, maintain the integrity of measurements, confidentiality and integrity of the attestation keys.
(C4) the channel between the `Attestation Client` and the `Monitor Module` is trusted.
(C5) The channel between the `Attestation Client` and the `Trust Module` is trusted.
(C6) The channel between the `Monitor Module` and the `Trust Module` is trusted.

**Implementation.** We can model the server system as a network system, and verify the server in a similar way as the network protocol verification. Specifically, we can 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 the attacker has the privilege to control the communication between two components, then we declare a public `network` for these two components. Otherwise if two modules are linked by one channel that is trusted, then we combine the two processes into one process so that the two modules can exchange messages directly.

**Results.** We consider and verify the sufficiency and necessity of the above preconditions that satisfy the security invariants. We use the same reachability functionality of ProVerif to verify the integrity property under different preconditions.

Precondition (C1) is not a necessary condition and can be removed from the TCB. ProVerif shows that the adversary cannot compromise the integrity of measurements, even if he takes control of the `Attestation Client`. If the adversary changes **VID** or **MR**

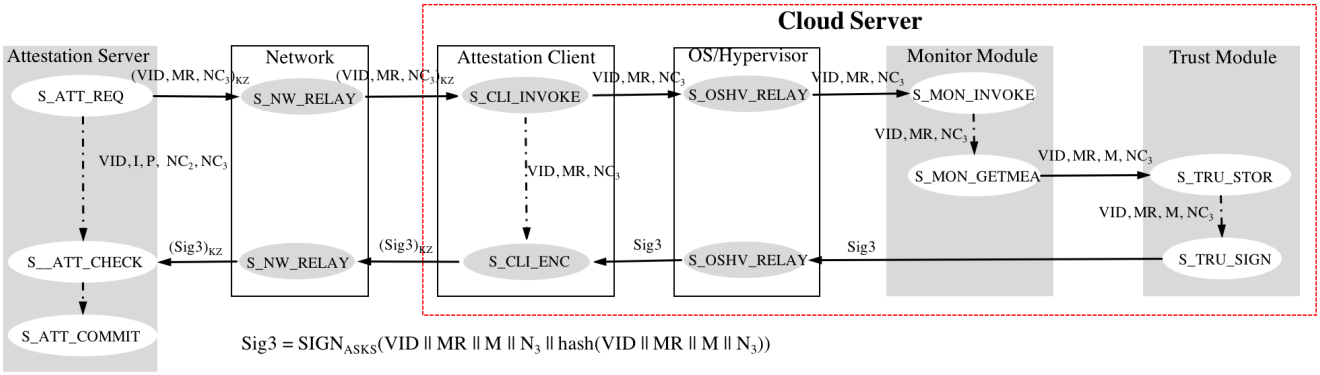


Figure 3: Internal protocol (interactions) in the cloud server. The gray blocks are trusted parties while the white blocks are untrusted parties. **KZ** is the symmetric key between the Attestation Server and the cloud server. **ASKS** is the private signing key of the cloud server. $\mathbf{NC_3}$ is the nonce used by the Attestation Server.

before they are sent to the `Monitor Module`, the `Monitor Module` will collect the wrong measurements **M**. However, the `Trust Module` will also sign the modified **VID** or **MR**, so the Attestation Server will notice this modification.

Precondition (C2) is a necessary condition to protect the measurements' integrity, and must be kept in the TCB. ProVerif also shows attack execution traces if this precondition is missing. For instance, if the `Monitor Module` is untrusted, it can collect wrong measurements **M**, and store them into the `Trust Module`.

Precondition (C3) is also necessary to guarantee the integrity of measurements. ProVerif shows the existence of attack execution traces without this condition. If the attacker obtains the attestation key in the `Trust Module`, it can generate a fake signature over any measurements using the signing key **ASKS**, while the Attestation Server can not detect this integrity breach. If the secure storage in the `Trust Module` is compromised, then an adversary can easily tamper with the security measurements without being detected by the Attestation Server.

Preconditions (C4) and (C5) are not necessary. The adversary cannot compromise the message integrity since the message in these channels is signed. Precondition (C6) is necessary. If this channel is not trusted, the adversary can modify the measurements, **M**, then the `Trust Module` will store and sign the wrong measurement.

Based on the above results, the necessary conditions to guarantee the measurements' integrity are: (1) the `Monitor Module` and the `Trust Module` and the communication channel between the `Monitor Module` and `Trust Module` must be trusted. ProVerif shows that it is sufficient for the cloud server to maintain the integrity of the measurements if only these subjects are included in the TCB of a cloud server.

The two modules should be correctly designed and implemented. We can verify a hardware module (e.g., `Trust Module`) in two steps. We can first check if the hardware design has any bugs, using well-known functional verification tools like SpyGlass [17] from Synopsys, HAL [18] from Cadence, etc. Then we can verify if the confidentiality and integrity of critical data are protected. We can use Gate-Level Information Flow Tracking (GLIFT) [19, 20, 21, 22] to formally verify these policies in the hardware design. Similarly, we can verify a software module (e.g., `Monitor Module`) in two steps. We can first check if there are software bugs and vulnerabilities (e.g., buffer overflow, memory leaks) in the implementation using static verification tools like Static Code Analyzer from HP Fortify Software [23], CodeSonar from GrammaTech [24], Secure Programming Lint [25], etc. Then we can use Information Flow Tracking [26, 27, 28] to verify if the security policies are enforced in the module.

## 4. VERIFICATION OF HYPERWALL

HyperWall [13] 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 no access to it. It is scrubbed by hardware before 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 TEC memory region. The TEC (Trust Evidence and Configuration) tables protect the memory of the guest VMs from accesses by the hypervisor or by DMA, depending on the customer's specification. Each memory region has associated entry in TEC tables specifying the access rights.

HyperWall can be used as the cloud server in the CloudMonatt, or it can be used stand-alone in a cloud computing scenario where there is remote user communicating to his or her (HyperWall) server located in the cloud possibly managed by untrusted cloud provider. HyperWall architecture is summarized in Figure 4 and below we present verification of one external protocol

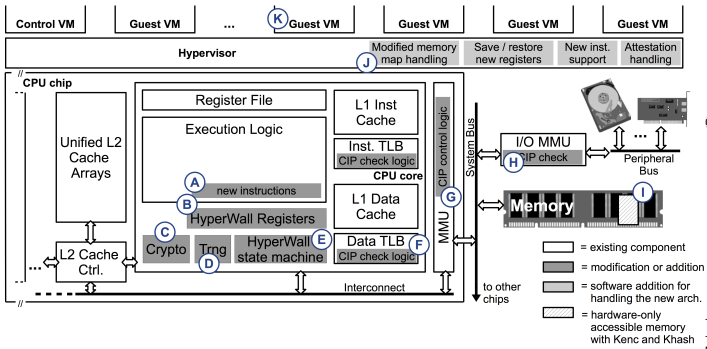


Figure 4: HyperWall added new instructions (Ⓐ), new registers (Ⓑ), a cryptographic engine (Ⓒ), a hardware random number generator (Ⓓ), core HyperWall state machine (Ⓔ); TLB logic (Ⓕ), the MMU (Ⓖ), and the I/O MMU (Ⓗ) were updated; portion of DRAM (Ⓘ) is used to store secure data and configuration, and the untrusted hypervisor (Ⓙ) and the guest VMs (Ⓚ) needs support for the new HyperWall instructions.

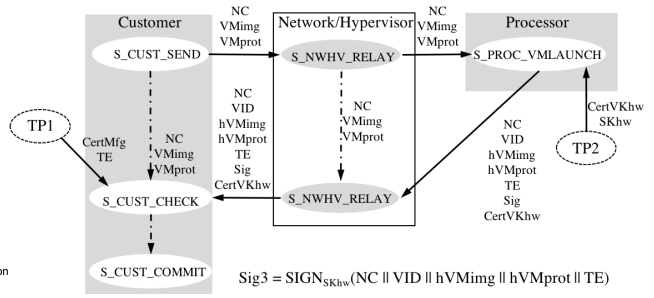


Figure 5: Model of VM Startup Validation external protocol. The gray blocks are trusted parties while the white blocks are untrusted parties. The key **SKhw** is the private key belonging to the processor, for which the customer has **CertMfg** certificate from manufacturer and the **CertVKhw** certificate of the **SKhw** sent by the `processor`, which is signed by the manufacturer.

and one internal interaction of HyperWall. We have further performed verification of five more HyperWall interactions, they are all summarized in Section 5.

## 4.1 External Protocol: VM Startup Validation

The security verification goal is to check if the untrusted network or hypervisor adversaries, between the customer and the remote server, can impact the integrity of the VM image and configurations requested.

**Modeling.** Figure 5 shows the external protocol with the involved components. The customer component "starts" VMs by specifying a nonce, **NC**, the virtual machine image **VMimg**, and the desired set of confidentiality and integrity protections for the virtual machine, **VMprot**. This "start VM" message is sent over the network to the hypervisor, which creates a data structure representing a VM. As the network and the hypervisor are potential attackers, the values sent could be altered (fabricate different values or replay old values). 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 assigns the VM identifier, **VID**, and signs – with its secret key **SKhw** – values that will define the VM: **NC**, **VID**, **hVMimg**, **hVMprot**, and **TE**. The **hVMimg** and **hVMprot** are the hash of the image of the started virtual machine and the hash of the requested protections, respectively. They are generated by the VM launch mechanisms invoked when the VM is launched. **TE** is the initial trust evidence, where initially the counters of memory access violations should be zero. The five values and their signature, **Sig**, and a certificate from the hardware manufacturer with the verification key needed to check the signature, **CertVKhw** are sent back to customer. **CertVKhw** 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 **CertMfg** and the initial expected value of **TE**. The processor knows the key, **SKhw** that it uses to make the signatures. It also has a certificate for the corresponding public key, **VKhw**, for recipients to verify its signatures. This information is made explicit as inputs from the two trusted party states, **TP1** and **TP2** and it is labeled `private`.

**Security invariants.** We identify one invariant:

① The customer is able to reach the commit state with **NC**, **VID**, **hVMimg**, **hVMprot**, **TE**, **Sig** and **CertVKhw** 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) Processor is trusted.
(C2) Processor has valid **CertVKhw** and **SKhw**.
(C3) Cloud user has valid **CertMfg** 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 `Network` principal with states corresponding to two points where the network needs to relay the data (and when the data 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 **NC**, **VID**, **hVMimg**, **hVMprot**, **TE**, **Sig** and **CertVKhw** is protected against fabrication of values and replay of values. **NC** and **TE** satisfy the case that there are known good values to compare against for these invariants. For **CertVKhw** there is

the **CertMfg** that can be used to compare against it and verify it, while **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.

The integrity of **hVMimg** and **hVMprot** is checked against fabrication and replay of values, since it is a hash primitive. The hashes are included in the signature, **Sig**, so they cannot be forged.

The integrity of **Sig** is checked against fabrication and replay of values: neither the network nor the hypervisor have access to the private signing key **SKhw** (hence they cannot fabricate the signature) and the customer has access to a chain of certificates that allows for him or her to verify the signature. For replay, the customer can check the nonce, **NC**, that he or she generated for this run of the protocol.

The (trusted) manufacturer's certificate, **CertMfg**, is used to check the server's certificate, **CertVKhw**; the server's certificate in turn is used to check the signatures. In particular, the **VID** is not known before the start of the protocol, but because it can be checked with the signature **Sig**, the value cannot be spoofed. This protocol also properly uses nonces as the **NC** generated by the customer in its initial state can later be compared to the received value.

## 4.2 Internal Interaction: VM Launch

We now show how the use of the Murphi model checker allows integrated functional and security verification of the state machine for setting up protections for the VM's pages.

**Modeling.** Figure 6 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 6:

(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 protection tables (CIP) are protected. (4) Finally 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 measurement of the VM is taken.

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

(C1) Processor is trusted.
(C2) Processor has valid **CertVKhw** and **SKhw**.

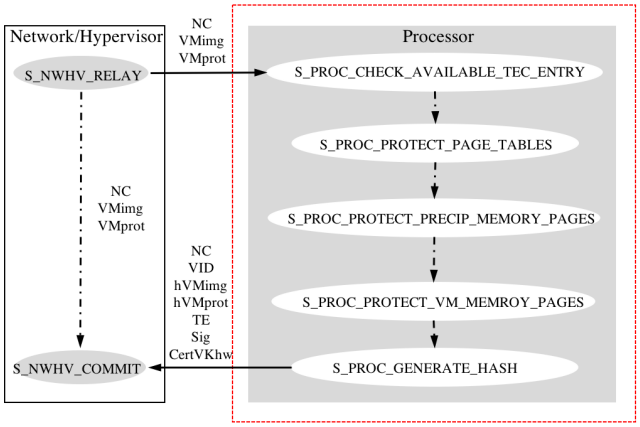


Figure 6: Model of VM Launch Mechanisms. The gray blocks are trusted parties while the white blocks are untrusted parties. The **CertVKhw** is the certificate of the signing key used by the `processor` in creating the signature $Sig$.

**Implementation.** As above, we model the untrusted network and untrusted hypervisor as a single `Network` 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: **NC**, **VID**, **hVMimg**, **hVMprot**, **TE**, **Sig** and **CertVKhw**. 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: **NC**, **VID**, **hVMimg**, **hVMprot**, **TE**, **Sig** and **CertVKhw**. 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.

# 5. VERIFICATION EVALUATION

In addition to the protocols presented in this paper, we have also verified two more for CloudMonatt and five more for HyperWall. For CloudMonatt we use ProVerif 1.88 with default options. 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.

The collected results for CloudMonatt are shown in Table 2 and for HyperWall in Table 3. The lines of code does include some comments which are very helpful for understanding the verification. The verification process is iterative, where the Murphi or ProVerif files may be updated many times, thus comments are crucial to understand the development of the verification strategy. We can also observe that the runtime time is also very small: due to the breakdown of internal and external verification, we can verify complex architectures within

Table 2: CloudMonatt verification evaluation results. First column shows what is being modeled, second column shows if it is internal interaction (Int.) or external protocol (Ext), third column shows the lines of code, and last column shows the runtime.

| Model | Int. or Ext. | Lines of Code | Run--time |
|---|---|---|---|
| External | Ext. | 262 | 0.2 |
| Cloud Server | Int. | 123 | 0.1 |
| Attestation Server | Int. | 205 | 0.2 |
| Cloud Controller | Int. | 187 | 0.1 |

Table 3: HyperWall verification evaluation results. Table columns are same as in Table 2.

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

a very short time. The most effort-consuming step is the design and writing of the verification models, but the actual verification is quick.

# 6. SECURITY DISCUSSION

While most of the protocols passed verification, the methodology did find potential design flaws and was used in improving the design of the two architectures.

## 6.1 Verification Impact on CloudMonatt

One of the most interesting results of security verification is to show how the Trusted Computing Base can be reduced. In CloudMonatt, it showed that only the Monitor Module and Trust Module of a cloud server should be included in the TCB, but the attestation client need not be trusted. Normally, third party customers only get guest VM privilege (ring 0) while the Monitor Module and Trust Module have hypervisor privilege (ring -1). So a normal tenant has no capability to subvert the security functions provided by these two modules. We note that the hypervisor is trusted in CloudMonatt's trust model (unlike for HyperWall), so software components in the hypervisor can be trusted. Secure enclaves can be used to protect the execution environment of the Monitor Module and Trust Module, leveraging mechanisms provided by Bastion [29, 30] or Intel SGX [8, 9]

## 6.2 Verification Impact on HyperWall

The verification effort uncovered two flaws in the initial Hyperwall design presented in [2], and later fixed in [13]. The first was a subtle replay attack in the VM Suspend and Resume protocol. The original design [2] had specified that a nonce provided by the customer would 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. Both were found when the invariants in the Murphi models did not pass.

# 7. RELATED WORK

**Secure architecture verification.** Past projects have looked at attempting to perform formal verification of an entire architecture [31]. Of the different architectures, XOM [32] and SecVisor [33] have received the benefit of verification using model-checking. XOM verification [3] checked read-protection and tamper-resistance against a highly simplified model of the instruction set operations on registers and memory, and SecVisor verification [34] used a logical system to reason about how security demonstrated by a small model of the SecVisor reference monitor can scale to the full implementation size. The IBM 4758 cryptographic co-processor's design included formal modeling and verification of the internal software [35].

**Security verification tools.** A number of tools specifically for security verification have been developed. Some tools focus on security protocols. They are designed to automatically verify the cryptographic protocols and find attacks that could undermine the security protocols which focus on secrecy and authentication. These tools include HERMES [36], Casrul [37], AVISPA [38], Scyther [39], ProVerif [11], Athena [40], etc.

Various model checkers have been used in security verification. These tools have no built-in security-related features and are meant for functional verification. But they include concepts such as invariants which can be adapted to define security invariants as well. Typical model checkers include Maude [41], Alloy [42], Murphi [12], CSP [43], FDR [44], etc.

In contrast to the above work, this paper proposes a new and general-purpose verification framework for secure architectures. We use two case studies to show that both the protocol verification tools (ProVerif) and model checkers (Murphi) can be used by this framework to implement the verification task. We believe other tools listed above can also be adopted by our framework to achieve this goal in a similar way.

# 8. CONCLUSION

In this paper, 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, deriving security invariants, and creating the implementation. We use two case studies to evaluate our methodology: security verification of a distributed cloud system, CloudMonatt, and verification of a secure processor architecture, HyperWall. Our case studies show that we can verify the design of complex secure architectures

efficiently. Moreover, they show that design bugs can be discovered, and fixed, thanks to this methodology. We envision our methodology can be easily adopted by computer architects to make their designs more secure.

## 9. ACKNOWLEDGMENTS

This work was supported in part by NSF 1526493.

## 10. REFERENCES

[1] T. Zhang and R. B. Lee, "CloudMonatt: An Architecture for Security Health Monitoring and Attestation of Virtual Machines in Cloud Computing," in *ACM International Symposium on Computer Architecture*, 2015.

[2] J. Szefer and R. B. Lee, "Architectural Support for Hypervisor-Secure Virtualization," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pp. 437–450, March 2012.

[3] D. Lie, J. C. Mitchell, C. A. Thekkath, and M. Horowitz, "Specifying and verifying hardware for tamper-resistant software," in *Proceedings of Symposium on Security and Privacy*, S&P, pp. 166 – 177, 2003.

[4] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th annual International Conference on Supercomputing*, ICS '03, pp. 160–171, 2003.

[5] R. B. Lee, P. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Proceedings of the International Symposium on Computer Architecture*, ISCA, pp. 2–13, 2005.

[6] 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*, CCS '07, pp. 389–400, 2007.

[7] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, HPCA, pp. 1–12, 2010.

[8] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *ACM International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[9] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *ACM International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[10] AMD, "AMD Memory Encryption." `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`, accessed May 2016.

[11] B. Blanchet, "Proverif automatic cryptographic protocol verifier user manual," *CNRS, Departement dInformatique, Ecole Normale Superieure, Paris*, 2005.

[12] D. L. Dill, "The murphi verification system," in *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV, pp. 390–393, 1996.

[13] J. Szefer, *Architectures for Secure Cloud Computing Servers*. PhD thesis, Princeton University, 2013.

[14] A. Datta, A. Derek, J. C. Mitchell, and A. Roy, "Protocol composition logic (pcl)," *Electronic Notes in Theoretical Computer Science*, vol. 172, no. 0, pp. 311–358, 2007.

[15] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *International Conference on Principles of Security and Trust*, 2012.

[16] T. Zhang and R. B. Lee, "Monitoring and Attestation of Virtual Machine Security Health in Cloud Computing," *IEEE Micro*, vol. 36, no. 5, 2016.

[17] "Spyglass lint: Early design analysis for logic designers." `https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-lint.html`.

[18] "Jaspergold automatic formal linting app." `https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-automatic-formal-linting-app.html`.

[19] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: A hardware description language for secure information flow," in *ACM Conference on Programming Language Design and Implementation*, 2011.

[20] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[21] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[22] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[23] H. P. Enterprise, "Fortify static code analyzer." `https://saas.hpe.com/en-us/software/sca`.

[24] GrammaTech, "Codesonar - static analysis sast software." `https://www.grammatech.com/products/codesonar`.

[25] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *USENIX Security Symposium*, 2001.

[26] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *ACM Symposium on Principles of Programming Languages*, 1999.

[27] V. Simonet and I. Rocquencourt, "Flow caml in a nutshell," in *Applied Semantics II workshop*, pp. 152–165, 2003.

[28] P. Li and S. Zdancewic, "Encoding information flow in haskell," in *IEEE Workshop on Computer Security Foundations*, 2006.

[29] D. Champagne and R. Lee, "Scalable architectural support for trusted software," in *Intl. Symp. on High Performance Computer Architecture*, 2010.

[30] D. Champagne, *Scalable Security Architecture for Trusted Software*. PhD thesis, Princeton University, 2010.

[31] A. DeHon, B. Karel, T. F. Knight, Jr., G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, and G. Sullivan, "Preliminary design of the SAFE platform," in *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS, pp. 1–5, 2011.

[32] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *SIGPLAN Not.*, vol. 35, pp. 168–177, November 2000.

[33] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 335–350, 2007.

[34] 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 *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, S&P, pp. 365–379, 2010.

[35] S. Smith, R. Perez, S. Weingart, and V. Austel, "Validating

a High-Performance, Programmable Secure Coprocessor," in *Proceedings of the 22nd National Information Systems Security Conference*, NISSC, October 1999.

[36] L. Bozga, Y. Lakhnech, and M. PÃl'rin, "HERMES: An Automatic Tool for Verification of Secrecy in Security Protocols," in *Computer Aided Verification* (W. A. H. Jr. and F. Somenzi, eds.), vol. 2725 of *Lecture Notes in Computer Science*, pp. 219–222, Springer Berlin Heidelberg, 2003.

[37] V. Cortier and B. Warinschi, "Computationally sound, automated proofs for security protocols," in *Programming Languages and Systems* (M. Sagiv, ed.), vol. 3444 of *Lecture Notes in Computer Science*, pp. 157–171, Springer Berlin Heidelberg, 2005.

[38] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, "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, pp. 281–285, 2005.

[39] C. J. Cremers, "The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols," in *Computer Aided Verification* (A. Gupta and S. Malik, eds.), vol. 5123 of *Lecture Notes in Computer Science*, pp. 414–418, Springer Berlin Heidelberg, 2008.

[40] D. X. Song, "Athena: A new efficient automatic checker for security protocol analysis," in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pp. 192–202, 1999.

[41] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The maude ltl model checker," *Electronic Notes in Theoretical Computer Science*, vol. 71, no. 0, pp. 162–187, 2004.

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

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

[44] A. W. Roscoe and Z. Wu, "Verifying statemate statecharts using csp and fdr," in *Formal Methods and Software Engineering* (Z. Liu and J. He, eds.), vol. 4260 of *Lecture Notes in Computer Science*, pp. 324–341, Springer Berlin Heidelberg, 2006.