Formal Modeling and Verification of CloudProxy

Wei Yang Tan[1], Rohit Sinha[1], John L. Manferdelli[2], and Sanjit A. Seshia[1]

[1] University of California, Berkeley, CA, USA
[2] Intel Science and Technology Center for Secure Computing

Abstract. Services running in the cloud face threats from several parties, including malicious clients, administrators, and external attackers. CloudProxy is a recently-proposed framework for secure deployment of cloud applications. In this work, we present the first formal model of CloudProxy, including a formal specification of desired security properties. We model CloudProxy as a transition system in the UCLID modeling language, using term-level abstraction. Our formal specification includes both safety and non-interference properties. We use induction to prove these properties, employing a back-end SMT-based verification engine. Further, we structure our proof as an "assurance case", showing how we decompose the proof into various lemmas, and listing all assumptions and axioms employed. We also perform some limited model validation to gain assurance that the formal model correctly captures behaviors of the implementation.

1 Introduction

With computation shifting to the cloud, security in cloud computing has become a concern. Providers of Infrastructure as a Service (IaaS) lease data center resources (processors, disk storage, etc.) to mutually non-trusting users. While IaaS providers use virtualization to isolate users on a physical machine, even if the virtualization software is assumed to be secure, a malicious user may still exploit misconfigurations or vulnerabilities in management software to gain complete control over data center networks and machines. Moreover, a malicious data center administrator can steal or modify unprotected disk storage. This can be catastrophic because applications may save persistent secrets (e.g. databases, cryptographic key) and virtual machine images (containing trusted program binaries) to disk. These threats are a challenge for deploying security-critical services to the cloud.

CloudProxy [16] is a recently-proposed framework for securely deploying cloud applications on commodity data center hardware. It implements a trusted service that is available to applications to 1) protect confidentiality and integrity of secrets stored on secondary storage, 2) cryptographically prove that they are running unmodified programs, and 3) securely communicate with other applications over untrusted networks.

We consider the problem of formal specification and verification of CloudProxy. We are concerned with proving that CloudProxy provides a set of security properties to any application that uses its API. To that end, we model the internals of CloudProxy in the presence of arbitrary, non-deterministic applications. Our first challenge is that the security guarantees listed above are informal and fairly high-level; it is non-trivial to formulate these properties for a detailed model of CloudProxy. Therefore, we construct an assurance case [20] that decomposes our proof into several axioms, assumptions about our trusted computing base, and lemmas that must be proved. The assurance case argues that our lemmas are complete — under our documented assumptions, our lemmas imply the high-level

security goals outlined by the authors of CloudProxy [16]. In formalizing these lemmas, we use well-known characterizations of non-interference [10] and semantic information flow [13]. Finally, we build a detailed term-level [5] model of CloudProxy, and prove these properties using a Satisfiability Modulo Theories (SMT) based theorem prover [3].

In summary, the primary contributions of this paper include:

- a formal model of CloudProxy (see Section 4)
- an assurance case for systematically decomposing our proof into a set of assumptions made by CloudProxy, and properties that must be proved on the model (see Section 3: Figure 2 and Table 3)
- a semi-automatic, machine-checked proof of our properties on the formal model (see Section 5)

2 Background

2.1 CloudProxy's Threat Model

We outline CloudProxy's threat model, which is described in greater detail in [16]. The adversary controls everything outside of the protected application's trusted computing base (TCB): hardware and OS/hypervisor that is running CloudProxy. That is, the adversary has physical access to all data center hardware and infrastructure, except the hardware (i.e. CPU, memory, chipset, backplane, disks) on which the protected application is currently running — there is no direct access to the hardware during operation and for a few minutes thereafter (to prevent cold boot attacks [12]). In practice, providers of Infrastructure as a Service (IaaS) may enclose racks of processors in cages to prevent physical access. However, a malicious administrator can remove, examine, modify the disk, and later re-install the modified disk on a CloudProxy machine. The adversary also controls all data center networks. In this threat model, CloudProxy protects the protected application's secrets that 1) reside locally on the machine, and 2) are communicated to other trusted applications over an untrusted network channel.

2.2 Overview of CloudProxy Architecture

Figure 1 gives a structural overview of the CloudProxy architecture. CloudProxy assumes that it runs on trusted hardware, which includes a trusted CPU, and a trusted motherboard containing a secure co-processor called the TPM [17]. The TPM serves as a root of trust for secure boot, cryptographic *sealing*/*unsealing* of secrets, and *attestation* of applications. *Sealing* encrypts the secret, and also binds it to the measurement of the application invoking the API; *unsealing* decrypts the sealed secret if and only if the measurement of the caller matches the bound measurement of the ciphertext. A measurement is a cryptographic hashing on the state of the entity of concern. The TPM protects the sealing keys within its hardware, thereby protecting the keys from software attacks. *Attestation* is a mechanism by which a remote party can verify that the local platform has a desired measurement, and then provision secrets to the local platform. The TPM-enabled boot eventually launches the operating system, which is also trusted by CloudProxy– section 3 describes what guarantees we require from a trusted OS. At the time of writing, CloudProxy uses a hardened Linux kernel. The crux of CloudProxy is the TCService process which exposes an API (see

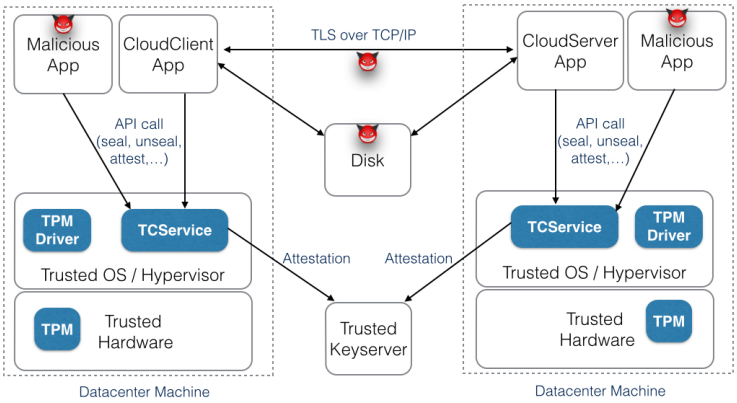

Fig. 1. Overview of CloudProxy architecture

Section 2.4) to its mutually trusting applications. The application uses this API to 1) $seal$ its secrets before saving them to disk storage, 2) $measure$ itself and the underlying OS to prove that it is running unmodified code, and 3) authenticate itself to remote CloudProxy applications via the $attest$ API.

We briefly describe how this architecture protects us from an adversary with capabilities as described in the threat model above. First, CloudProxy uses a trusted OS/hypervisor layer for isolating the protected application's execution from other adversarial applications. We argue that apart from vulnerabilities in the application logic (which is beyond our scope), the TCService API is the only remaining means of attack from adversaries. In this paper, we prove that TCService prevents any application's API request from interfering with another application's API response. Secondly, to protect from insider attacks that steal or modify disks, TCService provides a $seal$ (and corresponding $unseal$) API to add cryptographic confidentiality and integrity protection before writing secrets to disk. Since disks also store binaries within an application's TCB, TCService uses the TPM to measure the entire software stack (OS, TCService, CloudClient) before executing it. Lastly, to protect from attacks that observe or tamper messages sent over network, TCService provides an $attest$ API that an application can use to authenticate itself to a KeyServer. If the application has the expected measurement, TCService will return a certificate (signed by KeyServer) containing the application's public key. The application uses this certificate to establish a secure channel with another CloudProxy application over the network, thereby preventing network attacks. We use an assurance case in Section 3 to make a systematic argument for why CloudProxy provides sufficient defense against this threat model.

2.3 Deploying and Initializing CloudProxy

CloudProxy is deployed in two parts: 1) a virtual machine image containing the trusted OS and all CloudProxy applications, and 2) the trusted *KeyServer*. The *KeyServer* is deployed with the desired measurement of TCService, and desired measurements of each application. When the machine boots up and starts TCService, TCService uses the TPM to measure its trusted computing base (the OS and TCService binary), and sends a TPM's attestation to this measurement along with TCService's public key to the *KeyServer*. If the measurement matches the expected value, the *KeyServer* returns a certificate binding

TCService to its public key. This establishes trust between the *KeyServer* and TCService for all future communication. Next, TCService starts the application, e.g. CloudClient in Figure 1. To establish trust with the *KeyServer*, CloudClient uses TCService to measure its trusted computing base (the OS, TCService, and CloudClient binary), and sends the TCService's attestation to this measurement along with the CloudClient's public key to the KeyServer. In response, the *KeyServer* produces a signed certificate binding CloudClient to its public key. From hereon, CloudClient uses this certificate for establishing secure connections with other applications such as the CloudServer. The application also generates a private attestation key, which it $seals$ using TCService and saves to disk for future use. Note that the TPM acts as a hardware root of trust in this entire process.

2.4 CloudProxy API

Once the applications have been initialized, they may invoke any of the following CloudProxy API, in any order. We now briefly describe the semantics of each API function (details found in [16]).

1. *GetHostedMeasurement()*: computes the measurement of the calling application.
2. *Attest(data)*: returns a certificate (signed by TCService) binding *data* to the caller by including the caller's measurement in the certificate.
3. *GetAttestCertificate()*: returns a certificate (signed by KeyServer) binding the caller's public key.
4. *Seal(secret)*: encrypts the concatenation of *secret* and the caller's measurement. Then the message authentication code (MAC) of this ciphertext is attached to the ciphertext.
5. *Unseal(sealed_secret)*: performs integrity check on the MAC, and decrypts the input data if the integrity check succeeds. Next, TCService checks if the caller's measurement is equal to the measurement field in the plaintext. If this check succeeds, the plaintext is returned to the caller.
6. *GetEntropy(n)*: returns a cryptographically-strong random number of size *n* bits.

3 Assurance Case

We prove that CloudProxy protects its client applications from the threats allowed in our threat model. However, the description of the threat model and desired properties in the original CloudProxy paper are quite informal. Our first contribution in this work is to formalize these high-level security properties into a set of axioms, assumptions, and lemmas that are expressible within a model of CloudProxy. Although we formalize our assumptions and lemmas, we rely on an informal assurance case as a meta-level argument for why our lemmas and assumptions fulfill the high-level security properties. In Section 4, we build a formal model of CloudProxy, and in Section 5, we prove a set of lemmas on this model.

An assurance case is a documented body of evidence that provides a systematic, albeit informal, argument that a system satisfies a set of properties [20]. An assurance case first starts with a goal, and then iteratively decomposes it into constituent goals and assumptions, until all goals are supported by direct evidence. We follow the Goal Structuring Notation (GSN) as described in the GSN Community Standard [2]. A goal or a claim (marked by box labeled G) is a lemma we would like to prove. An assumption (marked by

oval labeled A) represents an assumption or an axiom in our proof. A context (labeled Ct) is used to limit scope of our work. An evidence (marked by circle labeled E) refers to a proof and is used to support a goal. We use circles with dashed lines to indicate proofs that are in progress.

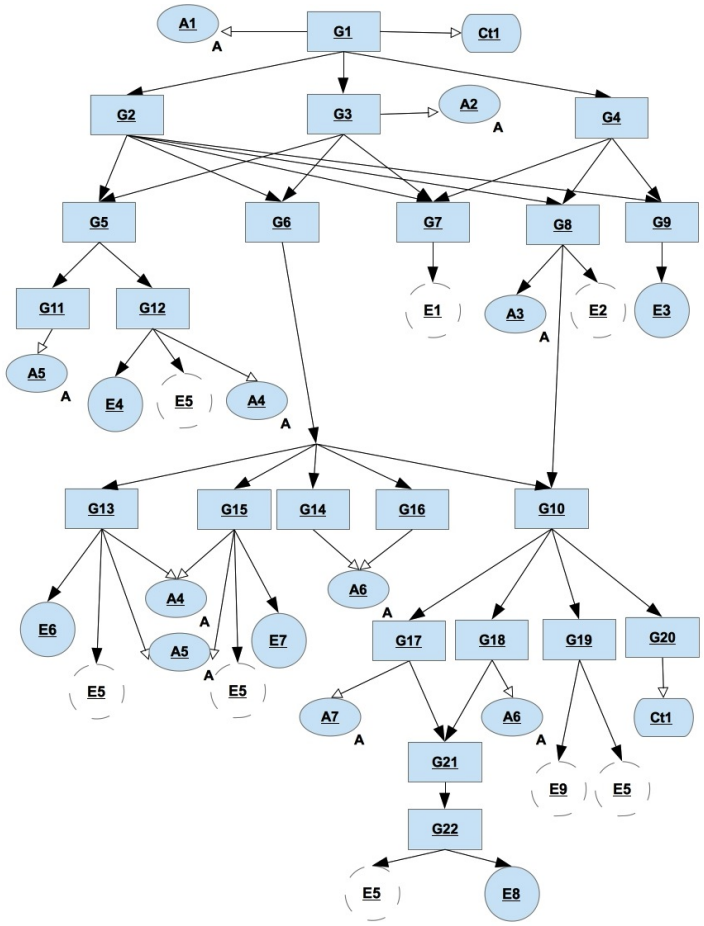

Fig. 2. CloudProxy assurance case.

As shown in Figure 1, CloudProxy relies on several components: a trusted hardware, a trusted OS/hypervisor layer, to-be-verified TCService, and a trusted remote key server. In this work, we only verify TCService, and assume that properties about other components hold. This is encoded as assumption **A1** in Figure 2: the hardware, the hypervisor, and the OS (including the TPM driver) are trusted.

For ease of exposition, we use the term "protected application" to refer to a CloudProxy application whose secrets we seek to protect, and the term "malicious application" to refer to any other CloudProxy application or program running on the same machine. Proving that CloudProxy protects the protected application's secrets (**G1**) is decomposed

Table 1. Node refers to the the assurance case node in Figure 2. **Proof Obligations** are either nodes in the assurance case, or property number(s) in Section 5.

Node	Description	Proof Obligation
A1	Hardware, hypervisor and OS are trusted.	
A2	Adversary cannot physically access computers that are running CloudProxy.	
A3	KeyServer is trusted.	
A4	Hypervisor and OS layers enforce separability.	
A5	Unique *pid* to all CloudProxy apps during TCService's lifetime.	
A6	Cryptographic primitives *seal*, *unseal*, SHA are implemented perfectly.	
A7	TPM driver does not leak TCService's secrets.	
Ct1	Verifying app logic is out of scope.	
E1	Verify measured launch mechanism.	
E2	Verify remote attestation protocol.	
E3	Use verified TLS implementation for network communication.	
E4	Prove **G12** on UCLID model.	Ppty (1), (2)
E5	Validate UCLID model.	
E6	Prove **G13** on UCLID model.	Ppty (6)
E7	Prove **G15** on UCLID model.	Ppty (7)
E8	Prove **G22** on UCLID model.	Ppty (9)
E9	Prove **G19** on UCLID model.	
G1	CloudProxy API secures protected app's secrets.	**A1**, **G2**-**G4**
G2	Secure against malicious programs running on same machine.	**G5**-**G9**
G3	Secure against malicious physical access of disk storage.	**A2**, **G5**-**G7**
G4	Secure against network attacks.	**G7**-**G9**
G5	Executions of any app do not affect other apps.	**G11**-**G12**
G6	Sealed secrets on disk have confidentiality and integrity protection.	**G10**,**G13**-**G16**
G7	Protected app and TCService are launched from unmodified code.	**E1**
G8	Remote attestation via untrusted channels.	**A3**, **E2**, **G10**
G9	Use TLS for establishing cryptographically secure channels.	**E3**
G10	Protected app and TCService do not reveal attestation and sealing keys.	**G17**-**G20**
G11	Isolation of address space belonging to TCService and apps.	**A5**
G12	Non-interference: Applications cannot affect each other through TCService API.	**A4**, **E4**-**E5**
G13	TCService *seal* API provides data confidentiality.	**A4**-**A5**, **E5**-**E6**
G14	Cryptographic library's *seal()* provides data confidentiality.	**A6**
G15	TCService *seal* API provides data integrity.	**A4**-**A5**, **E5**, **E7**
G16	Cryptographic library's *seal()* provides data integrity.	**A6**
G17	TCService does not reveal keys during initialization.	**A7**, **G21**
G18	Protected app does not reveal keys during initialization.	**A6**, **G21**
G19	TCService does not leak keys within responses to API calls.	**E5**, **E9**
G20	Protected app does not reveal keys after initialization.	
G21	CloudProxy initialization process does not reveal keys.	**G22**
G22	Arguments of system calls do not leak keys.	**E5**, **E8**

into 3 goals **G2** - **G4**, one for each ability granted to our adversary by the threat model. It must be noted that CloudProxy does not prevent an application from erroneously leaking

its secrets to the adversary; it only exports an API that, if used correctly, enables the application to protect its secrets. As a result, verifying application logic is out of scope (**Ct1**). Each goal in **G2** - **G4** is defined in terms of one or more goals in **G5** - **G9**. **G7** protects the application from attacks that change the application's binary or TCService's binaries on disk before the machine boots up. **G7** is supported by a proof of correctness of the measured launch sequence (**E1**), which uses the TPM to compute a cryptographic hash of the binaries before launching TCService and applications. We need not measure binaries after they launch because 1) we trust the OS/hypervisor to enforce memory protections, and 2) our threat model prevents an insider from physically accessing the memory chip of a machine running CloudProxy (**A2**). Note that all high-level goals **G2** - **G4** depend on **G7** because successfully mounting a compromised TCService binary will nullify all security guarantees. In addition to **G7**, we need **G5** and **G6** to guarantee **G2**: a protected application's secrets are not observable in plaintext by malicious programs on that machine. **G5** enforces that a malicious program does not observe a protected application's execution. Our notion of execution only considers application's state updates and side-effects via system calls; we do not consider information leaks via side channels. **G6** enforces that the protected application's secrets have cryptographic confidentiality and integrity protections before being saved to disk. **G8** and **G9** together protect an application's secret that is sent over the network. **G8** relies remote attestation to prove to a third-party that each protected application and TCService are running unmodified binaries. Following remote attestation, **G9** enforces that future communication takes place over a cryptographically secure channel. CloudProxy uses TLS (**E3**) for secure communication – we do not verify the TLS implementation in this work (this problem is explored in [4]).

Consider the assurance case for **G5**. This responsibility is shared between the OS protections (**G11**) and the TCService API guarantees (**G12**). **G11** stipulates that our OS 1) protects an application's address space from reads or writes by other programs, and 2) TCService is in full ownership of the TPM device. Both requirements can be fulfilled by a separation kernel [18]. While separability is a strict and possibly unreasonable requirement for commodity OS, for this discussion we assume we have a separation kernel via **A4**. As a result, TCService interface is the last remaining means by which a malicious application can interfere with the protected application's execution. To that end, **G12** stipulates a non-interference property on TCService: responses to the protected application's API requests is independent of the malicious application's API requests. We prove this property (**E4**) on our UCLID model, and make an initial attempt of validating this model with respect to the implementation (**E5**). Model validation proves that all behaviors in the implementation are captured by the model (see Section 6).

Consider the assurance case for **G6**. If secrets are sealed using TCService's *seal* API, then an adversary is unable to observe a secret's plaintext (confidentiality) and is also unable to tamper a secret's ciphertext without being detected (integrity). The proof for **G6** hinges on two sets of lemmas: 1) **G13-G16**: TCService's implementation of *seal* preserves confidentiality and integrity, and 2) **G10**: TCService never reveals its sealing key. For **G13-G16**, we assume (**A6**) that we have a Dolev-Yao [9] adversary — analyzing the strength of cryptographic operations is beyond our scope. In other words, our proof uses axioms of strong encryption, pre-image resistance of hash functions, second pre-image resistance, and strong collision resistance of hash functions. TCService performs *seal* by first encrypting the secret, and then appending the MAC (implemented using hash function) of the ciphertext. Goal **G14** is fulfilled by the confidentiality assumption about ideal encryption

scheme. Goal **G16** is fulfilled by the pre-image resistance, second pre-image resistance, and the strong collision resistance axiom about hash function used in MAC. We assume that the cryptographic library satisfies these axioms about encryption and hash functions. TCService also appends the application's measurement within the sealed secret. This measurement is used to decide if TCService should return the unsealed secret to the requester — the requester's measurement must match the measurement at the time of sealing. To that end, we also need goals **G13** (fulfilled by **E6**) and **G15** (fulfilled by **E7**) to prove that TCService does not *unseal* the protected application's secret on behalf of a malicious application. While building a formal model, we identified a design flaw (presented here as assumption **A5**) that the OS does not reuse process identifiers throughout the lifetime of TCService— TCService uses the process identifier (*pid*) to identify the application invoking the API call.

Consider the assurance case for **G10**. We must prove that this property holds during 1) TCService's initialization (**G17**), 2) application's initialization (**G18**), and 3) servicing of API request by TCService (**G19**). Although verifying application logic is out of scope, the application's initialization is handled by CloudProxy. **G18** proves that this initialization process does not leak keys. Both TCService and application use the same initialization routine, with the exception that the application uses the TCService's API for cryptographic operations, while TCService uses the TPM's API. This allows us to share **G21** for fulfilling both **G17** and **G18**. **E8** fulfills **G22** by proving that each write (e.g. file write, socket send) out of the application's address space is either sealed or the written value is independent of the keys. Finally, the proof in **E9** fulfills goal **G19**: TCService does not leak its sealing and attestation key in response to an API request. **G19** is necessary even after proving the non-interference property in **G12**. This is because TCService may leak the protected application's secrets by erroneously revealing its own sealing key.

4 Formal Modeling

Our assurance case in Section 3 allows us to focus our verification effort on the composition of TCService with the protected and malicious applications. Thus, we do not model the entire TCB consisting of the OS and hardware, since this TCB is not the focus of our verification effort. Instead, we use axioms and assumptions about the TCB in our model.

Figure 3 presents the structural overview of our model[3], for which we use the UCLID [5] modeling language. This model is a synchronous composition of four transition systems: 1) Protected application *App*, 2) Malicious application *Mal_App*, 3) *Scheduler*, and 4) TCService. The model captures the initialization routine (Section 2.3) of TCService and applications, as well as the semantics of each CloudProxy API. Recall that CloudProxy does not place any constraints on the application's behavior; secrets will get compromised if the application erroneously leaks the plaintext secrets or the private sealing keys. Therefore, we verify TCService in the presence of an arbitrary *App* and an arbitrary *Mal_App*. When triggered, *App* and *Mal_App* non-deterministically choose an API call and arguments to TCService in each step of execution. The *Scheduler* non-deterministically triggers either *App* or *Mal_App* to execute in each step. We choose interleaving semantics because the TCService implementation serializes all API requests onto a FIFO buffer, and handles each request atomically. Since *Mal_App* is completely non-deterministic, our proofs apply

[3] The model is available on http://uclid.eecs.berkeley.edu/cloudproxy

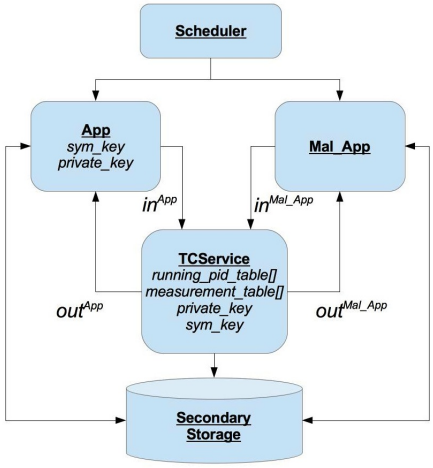

Fig. 3. UCLID model is a synchronous composition of *App*, *Mal_App*, *Scheduler*, and TCService.

to CloudProxy executions containing an unbounded number of malicious applications. TCService maintains the following state variables: 1) a private key (*private_key*) for remote attestation, 2) a symmetric key (*sym_key*) for use in $seal$ and $unseal$, 3) *running_pid_table[]* for process identifiers of all running CloudProxy applications, and 4) measurements *measurement_table[]* of all running CloudProxy applications. Each API operation may involve reading and writing to *Secondary_Storage*, which is modeled as an unbounded memory in the theory of Arrays.

As we are not analyzing the strength of cryptographic operations, we adopt the Dolev-Yao abstraction [9] in our model. Messages, keys, and state variables are modeled as terms. Cryptographic operations are uninterpreted functions over terms. The cryptographic operations are *perfect* — we apply axioms about strong encryption, pre-image resistance, second pre-image resistance, and strong collision resistance of hash functions.

The following lists the assumptions on the capabilities of *Mal_App* in our model:

1. *Mal_App* is able to execute any cryptographic operations as well as invoke any API of TCService.
2. At initial state, *Mal_App* does not have the knowledge of either *App* secrets or TCService keys in plaintext.
3. *Mal_App* is *not* able to eavesdrop on data returned by *TCService* to *App*. This assumption is sound since we assume that the OS is trusted, and the OS controls the response / request channel.
4. The malicious application has unlimited storage for data learned from invoking TCService APIs and cryptographic functions at every transition step. In other words, *Mal_App* may learn and generate new data from any combination of arbitrary function call.

During our modeling, we found a bug in the implementation. When a process terminates, the entry for that process *pid* is not removed from the *running_pid_table[]* and *measurement_table[]*. If the OS spawns a new application with the same *pid*, then the new application can start unsealing secrets belonging to the terminated CloudProxy application.

Having identified this bug, we introduce an assumption (**A5** in assurance case) that the *pid* will not be reused throughout the lifetime of TCService.

5 Verification

In this section, we formalize and verify properties on the UCLID model for each evidence in our assurance case. As mentioned previously, the evidences marked with a dashed line represent proofs that are currently in progress or left for future work. Each proof was performed using UCLID's internal decision procedures [5, 15].

5.1 Non-interference between Applications

G12 in Figure 2 stipulates that the responses to an application's API requests is independent of the malicious application's API requests. This means that *Mal_App*'s inputs to TCService can be removed without affecting TCService outputs to *App*, and vice versa. In the context of CloudProxy, this property requires two proofs:

1. **non-interference (secrecy)**: *App*'s secrets are not leaked to *Mal_App* when *Mal_App* invokes an API request
2. **non-interference (integrity)**: results of *App*'s API calls are unaffected by *Mal_App*'s API requests

We adopt Goguen and Meseguer's formalization of non-interference for both checks [10]. A trace is a sequence of states. Let T be the set of infinite traces allowed by the composition of $TCService \parallel App \parallel Mal\_App$. Also, let $in^{App}(t)$ and $in^{Mal\_App}(t)$ be the sequence of API requests invoked by *App* and *Mal_App*, respectively, in a trace t. Similarly, let $out^{App}(t)$ and $out^{Mal\_App}(t)$ be the sequence of API responses by TCService to *App* and *Mal_App*, respectively, in a trace t. The following property checks **non-interference (secrecy)** to *Mal_App*:

$$\forall t_1, t_2 \in T : (in^{App}(t_2) = \varepsilon \wedge in^{Mal\_App}(t_1) = in^{Mal\_App}(t_2)) \Rightarrow (out^{Mal\_App}(t_1) = out^{Mal\_App}(t_2)) \quad (1)$$

and the following property checks **non-interference (integrity)** from *Mal_App*'s API requests:

$$\forall t_1, t_3 \in T : (in^{Mal\_App}(t_3) = \varepsilon \wedge in^{App}(t_1) = in^{App}(t_3)) \Rightarrow (out^{App}(t_1) = out^{App}(t_3)) \quad (2)$$

where ε denotes no API invocation (modeled as stuttering steps). Note that this definition only applies if two conditions are met: 1) TCService must be deterministic (*App* and *Mal_App* need not be deterministic), and 2) TCService's transition function must be total with respect to inputs.

A hyperproperty is a set of sets of possibly infinite execution traces [7]. As properties (1) and (2) reason over a pair of traces, they are both are hyperproperties. We can rewrite them as 2-safety properties [7] and prove them using induction. As Figure 5(a) illustrates, we first construct a 2-fold parallel self-composition of the system, resulting in two

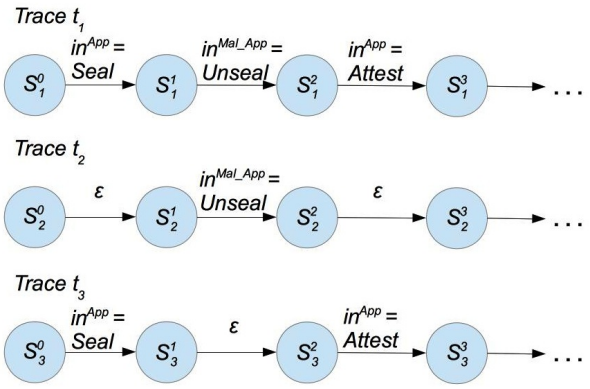

Fig. 4. The figure shows three traces t_1, t_2 and t_3, where trace t_2 replaces *App* API requests in t_1 with ε, and t_3 replaces *Mal_App* API requests in t_1 with ε.

instances Sys_1 and Sys_2 of TCService that run synchronously and use the same transition relation R. Let s_1 and s_2 be the state of TCService in Sys_1 and Sys_2 respectively. Let in_1 and in_2 be the input to TCService in Sys_1 and Sys_2 respectively. We also let in_n^{App} and $in_n^{Mal\_App}$ be to *App*'s input and *Mal_App*'s input to TCService in Sys_n respectively. Similarly, let $out^{App}(s)$ and $out^{Mal\_App}(s)$ refer to TCService's output in state s to *App* and *Mal_App* respectively. For **non-interference (secrecy)**, we prove the following inductive property:

$$\forall s_1, s_2. Init(s_1) \wedge Init(s_2) \Rightarrow \Phi_{\textit{Mal\_App}}(s_1, s_2) \tag{3}$$

$$\begin{aligned}&\forall s_1, s_1', s_2, s_2', in_1, in_2.\\ &(\Phi_{\textit{Mal\_App}}(s_1, s_2) \wedge R(s_1, in_1, s_1') \wedge R(s_2, in_2, s_2') \wedge in_2^{App} = \varepsilon \wedge in_1^{Mal\_App} = in_2^{Mal\_App})\\ &\Rightarrow \Phi_{\textit{Mal\_App}}(s_1', s_2')\end{aligned} \tag{4}$$

where

$$\begin{aligned}\Phi_{Mal\_App}(s_a, s_b) \doteq &\forall s_a', s_b', in.\\ &R(s_a, in, s_a') \wedge R(s_b, in, s_b') \Rightarrow (out^{Mal\_App}(s_a') = out^{Mal\_App}(s_b'))\end{aligned} \tag{5}$$

For any pair of states s_a and s_b, predicate $\Phi_{Mal\_App}(s_a, s_b)$ is $true$ if and only if those states are indistinguishable to *Mal_App* — for the same API call, TCService produces identical output in both s_a and s_b. We also use a transition predicate $R(s, i, s')$ which is $true$ iff the system can transition from state s to s' under input i. Property 3 checks the base case that $\Phi_{\textit{Mal\_App}}$ holds on any pair of initial states. The inductive step (property 4) proves that from any pair of states s_1 and s_2 that is indistinguishable to *Mal_App*, TCService must transition to a pair of states s_1' and s_2' (respectively) that are also indistinguishable to *Mal_App*. We also need an auxiliary invariant to discharge the induction proof: if the *App*'s *pid* entries of the measurement table in TCService in s_1 and s_2 are the same, then these entries have the same values in s_1' and s_2'.

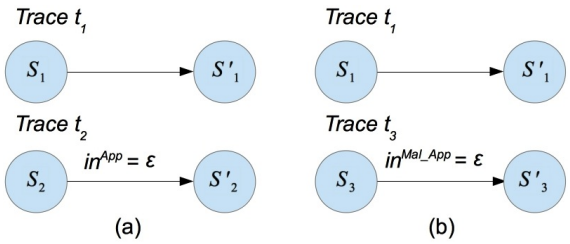

Fig. 5. S denotes the state of TCService in our UCLID model. We prove non-interference (secrecy) in (a) by proving that Mal_App cannot distinguish s'_1 from s'_2. We prove non-interference (integrity) in (b) by proving that App cannot distinguish s'_1 from s'_3.

Proving **non-interference (integrity)** between *App* and *Mal_App* requires a similar inductive proof – the preceding discussion applies verbatim if *App* is substituted for *Mal_App* and vice versa. UCLID took about 5 seconds to prove each property. [4]

5.2 Data Confidentiality

Here, we describe our proof of **G7**: *Mal_App* cannot acquire the plaintext of a sealed secret belonging to *App*. Recall from Figure 2 that we split this goal into two lemmas:

- **Lemma 1**: *Mal_App* cannot obtain the plaintext by breaking the underlying cryptography (goal **G14** in Figure 2).
- **Lemma 2**: *Mal_App* cannot obtain the plaintext by invoking a sequence of CloudProxy API calls (goal **G13** in Figure 2).

Lemma 1 is simply assumed in our work since we assume a Dolev-Yao adversary [9]. In accordance with the Dolev-Yao model, our model represents data as terms of some abstract algebra, and cryptographic primitives operate on those terms to produce new terms. TCService satisfies **Lemma 2** by appending measurement to the secret prior to sealing. During unsealing, if the secret's measurement does not match the measurement of API requester, then the request fails. In what follows, we prove that TCService implements this logic correctly.

Let m be a measurement, m_{App} be the *App*'s measurement, and $\mathbb{D}$ be the set of terms from an abstract algebra. Also, let ENC_MAC be the authenticated encryption function that first encrypts the plaintext, and then appends an integrity-protecting MAC of the ciphertext. Let $in_{API}^{Mal\_App}$ be the API call from the *Mal_App* to TCService, and let $in_{arg}^{Mal\_App}$ be the arguments of the API call from the *Mal_App* to TCService. $out_{success}^{Mal\_App}(s)$ denotes whether TCService successfully performed the API request invoked by Mal_App. $out_{result}^{Mal\_App}(s)$ is the return output of TCService to the *Mal_App*. sK_{TCS} denotes the symmetric key used by TCService to $seal$ or $unseal$. We define **Lemma 2** as follows.

$$\begin{aligned}\phi(s) \doteq & \forall secret \in \mathbb{D}, s', m_{App}, in. \\ & (in_{API}^{Mal\_App} = unseal \wedge in_{arg}^{Mal\_App} = ENC\_MAC(sK_{TCS}, secret, m_{App})) \wedge \\ & R(s, in^{Mal\_App}, s') \Rightarrow \neg out_{success}^{App}(s') \end{aligned} \quad (6)$$

[4] UCLID was running on VirtualBox and the machine was a 2.6GHz quad-core with 2GB of memory space allocated to this VirtualBox environment.

where $ENC\_MAC(sK_{TCS}, secret, m_{App})$ is a term encoding any arbitrary sealed secret that can belong to *App*, as $secret$ is an unconstrained symbolic constant. This allows us to only consider API calls whose argument has this form. This property guarantees that TCService never returns the plaintext secret as a result of calling $unseal$ API. **Lemma 1** guarantees that the adversary cannot obtain the plaintext from a sealed secret by breaking the underlying cryptography.

We prove **Lemma 2** via 1-step induction. UCLID took about 30 seconds to prove this property. Moreover, we discovered the following necessary assumption to prevent spurious counter-examples to the inductive proof: *Mal_App* has a different measurement than *App*, i.e. $m_{Mal\_App} \neq m_{App}$. This is reasonable because they run different binaries, and hash functions are assumed to be collision free.

5.3 Data Integrity

Similar to confidentiality, we prove that TCService enforces integrity protection: the adversary cannot tamper a sealed secret and still have TCService successfully $unseal$ it on behalf of *App*. Again, we assume perfect integrity protection of $ENC\_MAC(key, ., .)$, and hence any modification to $ENC\_MAC(key, ., .)$ should *not* be able to $unseal$ successfully. Only data that was previously sealed by TCService can be successfully unsealed by TCService– any other data would fail the MAC check since the MAC check uses TCService's symmetric key sK_{TCS}. This leaves the adversary with only one attack: replace *App*'s sealed data with *Mal_App*'s sealed data. Therefore, the following property checks that TCService does not $unseal$ another application's sealed data on behalf of *App*.

Let $\mathbb{M}$ be the set of measurements, and $\mathbb{D}$ be the set of data. We prove that an $unseal$ request satisfies the property:

$$\begin{aligned}\phi(s) \doteq & \forall secret \in \mathbb{D}, \forall m \in \mathbb{M}, s', in. \\ & in_{API}^{App} = unseal \wedge in_{arg}^{App} = ENC\_MAC(sK_{TCS}, secret, m) \\ & \wedge m \neq m_{App} \wedge R(s, in^{App}, s') \Rightarrow \neg out_{success}^{App}(s')\end{aligned} \tag{7}$$

where $ENC\_MAC(sK_{TCS}, secret, m)$ is a term encoding any sealed secret that can belong to an application other than *App*, as $secret$ and m are unconstrained symbolic constants.

UCLID took less than 5 seconds to prove this property. A caveat to note here is that CloudProxy does not currently have a mechanism to check for the *freshness* of data. The adversary may perform a replay attack by replacing the *App*'s sealed secret on disk with an older secret sealed by the *App*.

5.4 Protecting Keys

During initialization, TCService generates a symmetric sealing key sK_{TCS}, and a private attestation key pK_{TCS}. Similarly, a CloudProxy application uses TCService to generate a symmetric key sK_{App} and private attestation key pK_{App}. In this section, we prove that keys sK_{App} and pK_{App} are never leaked in writes outsides the App's address space (goal **G18**). We only focus our attention on *App*'s keys in this section; the property and proof for TCService is identical. We defer proof for TCService as it uses the same initialization routine as the application. We express this property in the semantic information flow

framework introduced by [13]. For any pair of traces, where the traces start from symbolic states differing in values of sK_{App} and pK_{App} (but all other state variables are identical), the unencrypted outputs along the two traces must be identical – the keys will affect the values of encrypted data. In other words, values written to disk are not a function of the keys. Once again, this is a 2-safety property of $TCService \parallel App \parallel Mal\_App$. We use a 1-step induction to prove this property.

First, we define a specification state variable $\mathcal{S}$ that gets updated each time *App* invokes TCService $seal$ API on some data or during initialization.

$$\mathsf{next}(\mathcal{S}(x)) = \begin{cases} true & in_{API}^{App} = seal \wedge x = ENC\_MAC(sK_{TCS}, in_{arg}^{App}, m_{App}) \\ true & init^{App} = true \wedge x = ENC\_MAC(sK_{App}, pK_{App}, m_{App}) \\ S(x) & \text{otherwise} \end{cases} \tag{8}$$

where $init^{App}$ is a boolean value that indicates whether *App* is at the initialization phase. In addition, $\forall x.S_0(x) = false$ where S_0 is the initial state of S.

Let s_1 and s_2 be a pair of states, where $pK_{App,1}$ and $pK_{App,2}$ are *App*'s private keys in s_1 and s_2 respectively. $sK_{App,1}$ and $sK_{App,2}$ are *App*'s symmetric keys in s_1 and s_2 respectively. $s_1 \setminus \{pK_{App,1}, sK_{App,1}\}$ denotes the set of all state variables in s_1 excluding the two keys. Finally, $out^{disk}(s_1)$ denotes the output to disk in state s_1, and $out^{disk}(s_2)$ denotes the output to disk in state s_2. We formulate this property as follows:

$$\begin{aligned} \forall s_1, s_2, s_1', s_2', in. \\ & (s_1 \setminus \{pK_{App,1}, sK_{App,1}\}) = (s_2 \setminus \{pK_{App,2}, sK_{App,2}\}) \\ & \wedge R(s_1, in^{App}, s_1') \wedge R(s_2, in^{App}, s_2') \\ & \wedge (\neg \mathcal{S}(out^{disk,1}(s_1')) \vee \neg \mathcal{S}(out^{disk,2}(s_2'))) \Rightarrow \\ & (out^{disk}(s_1') = out^{disk}(s_2')) \end{aligned} \tag{9}$$

UCLID took about two seconds to prove this property. An important caveat is that we only prove this property for writes that the CloudProxy initialization code of *App* makes via the system call interface (e.g. file write to disk). The soundness of this proof relies on the model validation proof that we have captured all possible writes to disk in our model.

6 Model Validation

Although we have proved the security properties of CloudProxy on the formal UCLID model, we are left with an important question: is the model a sound abstraction of the original system? A valid model must encode all behaviors that are allowed in the original system. We make first steps in using KLEE [6] to validate our UCLID model against the C++ implementation, using techniques proposed by Sturton et al. [21].

Since we do not precisely model all computation within TCService (e.g. messages are abstracted away as terms), we need to show that the unmodeled code does not affect the subset of TCService state that we have modeled. Let $\mathcal{V}$ denote the state variables that are present in our UCLID model. We manually identify code paths that will be *pruned* away from our modeling. Then, we prove that the *pruned* code does not affect any state variable

within $\mathcal{V}$. This proof uses the *Data-Centric Model Validation* (DMV) technique from [21]. Once we have validated our pruning, we must further prove that the model correctly abstracts the pruned program. This is termed as *Operation-Centric Model Validation* (OMV) in [21]. Both validation steps are a work in progress.

The entire TCService has about 58k lines of code (LoC), of which about 8k LoC is used to build our model. The cryptographic keys, measurement table, and the *pid* table in TCService are our $\mathcal{V}$ set, and only approximately 1k LoC modifies $\mathcal{V}$. After the DMV step, we model in UCLID the remaining 1K LoC. We encountered several challenges in performing OMV, and delay that to future work.

7 Related Work

There has been some use of formal methods for building trustworthy cloud infrastructure. CertiKOS [11] is a verified hypervisor architecture that ensures correct information flow between different guest users. They use a compositional proof technique to decompose their proof into individual lemmas that can be proved using different proof engines. Klein et al. [14] provide a machine-checked verification of the seL4 microkernel in Isabelle. These efforts are especially relevant since CloudProxy needs a trusted OS/hypervisor layer. While both efforts use interactive theorem proving for building machine checked proofs, we use a more automated methodology based on model checking. Another alternative approach is to directly prove the implementation code by inserting annotations and assertions, and then run a verifier on the code. VCC has been developed to verify the Hyper-V implementation using this approach [8]. More importantly, a carefully constructed model can raise the level of abstraction enough to prove such properties efficiently.

We structure our proof of correctness as an assurance case. Assurance cases have been applied in practice to present the support for claims about properties or behaviors of a system. ASCAD [1] presents safety cases (a slight variant of assurance case) for safety critical systems such as military systems. Shankar et al. [19] use Evidential Tool Bus to construct claims, and to integrate different formal tools to provide evidence for each claim.

8 Conclusion

We present the first formal model of CloudProxy, and an assurance case to systematically construct a proof that CloudProxy protects an application's secrets in our threat model. The assurance case lists practical assumptions we make about the trusted computing base of CloudProxy applications. During our modeling and verification of CloudProxy, we have uncovered a flaw and few unintended assumptions in the design (e.g. no reuse of *pid* during TCService's lifetime). Security properties and lemmas derived from the assurance case (e.g. non-interference) are formalized and proved in our model. In ongoing work, we are exploring a model validation technique to prove that our model encodes all the behaviors allowed by CloudProxy's implementation.

Acknowledgments We sincerely thank David Wagner and Petros Maniatis for their valuable feedback. This work was funded in part by the Intel Science and Technology Center for Secure Computing, and SRC contract 2460.001.

References

1. *Adelard: ASCAD The Adelard Safety Case Development (ASCAD) Manual.*, 1998.
2. *GSN Community Standard Version 1*, 11 2011.
3. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, Feb. 2009.
4. K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu. Verified cryptographic implementations for tls. *ACM Trans. Inf. Syst. Secur.*, 15(1):3:1–3:32, Mar. 2012.
5. R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. 2002.
6. C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI'08, pages 209–224, Berkeley, CA, USA, 2008.
7. M. Clarkson and F. Schneider. Hyperproperties. In *Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st*, pages 51–65, 2008.
8. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*.
9. D. Dolev and A. C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.
10. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
11. L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. Certikos: A certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 3:1–3:5, New York, NY, USA, 2011. ACM.
12. J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, may 2009.
13. R. Joshi and K. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1 - 3):113 – 138, 2000.
14. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Symposium On Operating Systems Principles*, pages 207–220. ACM, 2009.
15. S. Lahiri and S. Seshia. The uclid decision procedure. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478. Springer Berlin Heidelberg, 2004.
16. J. Manferdelli, T. Roeder, and F. Schneider. The cloudproxy tao for trusted computing. Technical Report UCB/EECS-2013-135, University of California, Berkeley, 07 2013.
17. B. Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, pages 9:1–9:6, Berkeley, CA, USA, 2008.
18. J. Rushby. Proof of Separability—A verification technique for a class of security kernels. In *Proc. 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 352–367, Turin, Italy, Apr. 1982. Springer-Verlag.
19. N. Shankar. Building assurance cases with the evidential tool bus. http://chess.eecs.berkeley.edu/pubs/1061.html, 03 2014.
20. J. Stephen Blanchette. Assurance cases for design analysis of complex system of systems software. Technical report, Software Engineering Institute, Carnegie Mellon University, 04 2009.
21. C. Sturton, R. Sinha, T. H. Dang, S. Jain, M. McCoyd, W.-Y. Tan, P. Maniatis, S. A. Seshia, and D. Wagner. Symbolic software model validation. In *Proceedings of the 10th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, 10 2013.