Formal Modeling and Verification of CloudProxy

Wei Yang Tan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-112
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-112.html

May 16, 2014

Formal Modeling and Verification of CloudProxy

by

Wei Yang Tan

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor David A. Wagner
Dr John L. Manferdelli

Spring 2014

The thesis of Wei Yang Tan, titled Formal Modeling and Verification of CloudProxy, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

Formal Modeling and Verification of CloudProxy

Abstract

Formal Modeling and Verification of CloudProxy

by

Wei Yang Tan

Master of Science in Computer Science

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

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 thesis, 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.

To my dad, my mum, my sis, and Xintong.

Contents

Contents **ii**

List of Figures **iv**

List of Tables **v**

1 Introduction **1**
1.1 Summary of Contributions 2
1.2 Related Work 2

2 Overview of CloudProxy **4**
2.1 CloudProxy's Threat Model 4
2.2 Overview of CloudProxy Architecture 5
2.3 Deploying and Initializing CloudProxy Applications 7
2.4 CloudProxy API 9

3 Assurance Case For CloudProxy **10**
3.1 Goal Structuring Notation (GSN) 10
3.2 Structuring CloudProxy Assurance Case in GSN 11

4 CloudProxy Abstraction **17**
4.1 Modeling in UCLID 17
4.2 Capabilities of *Mal_App* 19
4.3 Model Assumptions and Axioms 20

5 Verification **22**
5.1 Property 1: Non-interference 22
5.2 Property 2: Data Confidentiality 26
5.3 Property 3: Data Integrity 27
5.4 Property 4: Protecting Keys 28

6 Model Validation **30**

7 Conclusion **32**
7.1 Ongoing and Future Work 32

Bibliography **34**

List of Figures

2.1 Overview of CloudProxy architecture. 5
2.2 Sealing / unsealing keys at initialization. 7
2.3 Remote attestation of a CloudProxy application. 8

3.1 Elements in GSN. 12
3.2 Relationships among elements in GSN. 12
3.3 CloudProxy assurance case. 13

4.1 CloudProxy model in UCLID. 18

5.1 Non-interference property for CloudProxy. 23
5.2 Proving non-interference in UCLID. 24

List of Tables

3.1 Descriptions of assurance case nodes. 16

Acknowledgments

I would like to thank my advisor, Sanjit A. Seshia, for his guidance and patience, His enthusiasm and brilliant mentorship are inspiring. Though it has only been just about 4 semesters, I am glad to have the opportunity to work with Sanjit.

I would also like to thank David Wagner and John Manferdelli for being my thesis committee, as well as providing valuable feedback and insights for this work. In addition, I would like to thank Petros Maniatis for his feedback and insights.

This work could never be successful without Rohit Sinha. Being one of the main collaborators (and my *unofficial* senior), he has contributed a lot to this work, including UCLID modeling, model validation, and especially on deriving the properties and verifying them.

Special thanks to the *learn-and-verify* team: Wenchao Li, Alex Donze, Indranil Saha, Dorsa Sadigh, Ankush Desai, Daniel Fremont, Jonathan Kotker, Nishant Totla, Garvit Juniwal, Eric Kim, Matthew Fong (and of course Sanjit and Rohit), for being so patient with me, and for making the entire working environment so lively and conducive for research. I would like thank all my great friends from the DOP center: Antonio Iannopollo, Ho Yen-Sheng, Hokeun Kim, Pierluigi Nuzzo, Ben Zhang, Nikunj Bajaj, and Shromona Ghosh. Besides being my buddies for meals and travels, they have been providing great support and have always been there to help me.

Thanks to Ana Reyes for her help. I would like to thank my boss from DSO, Tan Yang Meng, and Lee Aik Tuan, for helping me realize my dream of furthering my studies. Special thanks to my DSO colleagues: Keegan Lim, Lim Kai Ching, Tan Jiaqi, Koh Ming Yang, Cho Chia Yuan for their support.

I am very grateful to Ong Yi Xiong for always motivating me and keeping my sanity. I am also very grateful to Leow Shi Chi for always cheering on me, and giving me great research advices.

Lastly, I would like to thank my dad, my mum and my sis for being so supportive. They are always there for me, and to lend me a listening ear.

The work described in this thesis was funded in part by the Intel Science and Technology Center for Secure Computing.

Chapter 1

Introduction

With computation steadily 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 tamper with unprotected disk storage. This can be catastrophic because applications may save persistent secrets (for example databases, cryptographic keys) and virtual machine images (containing trusted program binaries) to disk. These threats are a challenge for deploying security-critical services to the cloud.

CloudProxy [20] is a framework that is recently proposed for secure deployment of cloud applications on commodity data center hardware. It implements a trusted service that is available via an API 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.

In this thesis, we consider the problem of formal specification and verification of CloudProxy. Through formal verification, we aim to achieve higher assurance in CloudProxy for industrial adoption. Our first challenge is formulate security properties for a detailed model of CloudProxy. We construct an *assurance case* [22] that decomposes our proof into several axioms and assumptions about our trusted computing base, as well as lemmas that must be proved. This *assurance case* argues that our set of lemmas is complete — under our documented assumptions, our lemmas imply the high-level security goals outlined by the authors of CloudProxy [20]. Among many other lemmas, we prove that CloudProxy does not leak information between mutually non-trusting applications, or allow applications to interfere with each other via the CloudProxy API. In formalizing these lemmas, we use well-known

characterizations of non-inteference [10] and semantic information flow [19]. Finally, we build a detailed *term-level* [6] model of CloudProxy, and prove these properties using Satisfiability Modulo Theories (SMT) solver.

Term-level abstraction is a modeling technique which data is represented using symbolic terms, and precise functionality is abstracted away with uninterpreted functions [5, 6]. This is useful for our work because in several instances, we need to deal with data of arbitrary length (for example modeling binaries and data for encryption). Moreover, we can efficiently abstract cryptographic functions using uninterpreted functions, since we are not reasoning about the strength of the cryptographic primitives.

The SMT problem is a decision problem over first-order logic with (typically) background theories such as theory of equality and arrays [5, 4]. As compared to boolean satisfiability problem (SAT), SMT allows greater expressiveness through the use of first-order logic. This enables us to prove our properties on term-level models.

The structure of this thesis is as follows: first we give an overview of CloudProxy. Then we describe our argument through assurance case to derive the security properties, the assumptions, as well as the lemmas. Next, we give a brief description of the CloudProxy model. Lastly we elaborate on the security properties that we have formulated and how we perform verification on the model.

1.1 Summary of Contributions

The primary contributions of this thesis include:

- a formal model of CloudProxy (Chapter 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 (Chapter 3)
- a semi-automatic, machine-checked proof of our properties on the formal model (Chapter 5)

We begin in Chapter 2 with a brief description of CloudProxy.

1.2 Related Work

There has been some use of formal methods for building trustworthy cloud infrastructure. CertiKOS [12] 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. On that note, Klein et al. [17] provide a machine-checked verification of the seL4 microkernel in Isabelle. These efforts are especially interesting since CloudProxy relies on 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.

Our work builds upon several notions of secure computation proposed in literature. For instance, we find applications of non-interference proposed by Goguen and Meseguer [10]. We also use the notion of semantic information flow proposed by Joshi et al [19].

Assurance cases have been applied in practice to present the support for claims about properties or behaviors of a system. [1] presents safety cases (a slight variant of assurance case) for safety critical systems such as military systems. Shankar et al. [24] use Evidential Tool Bus to construct claims, and to integrate different formal tools to provide evidence for each claim.

Chapter 2

Overview of CloudProxy

CloudProxy is a software framework that implements secure, distributed, cloud-based services. It is implemented as a stack of layers: the Trusted Hardware (TrHW), the Trusted Hypervisor (TrHV), the Trusted Operating System (TrOS), and applications running on top of TrOS. The Tao is part of the CloudProxy framework that enables recursive TrHW, TrHV and TrOS form part of the trusted computing base for an application. Each layer consists of a hosted system with CloudProxy services provided by a host; for example, the TrHW is the host for the TrHV, and TrHV is the hosted system of TrHW. We label these applications as CloudProxy applications or activity elements, and together they form an activity. An activity is an instance of a distributed computation executing on behalf of some activity owner. These CloudProxy applications use the services of CloudProxy to protect their secrets.

In this chapter, we will first look at the threat model CloudProxy defends against (Section 2.1). We will then cover the core aspects of the CloudProxy architecture and implementation, which are involved in our formal verification, in Section 2.2.

2.1 CloudProxy's Threat Model

We briefly describe CloudProxy's threat model. The scenario consists of data center machines leased to mutually untrusting users and managed by possibly malicious data center administrators. A malicious client can exploit vulnerabilities in data center software to assume control of all machines (except the machines running CloudProxy), as well as the networks in the data center. A malicious administrator can move, examine, modify the disk, and later re-install the modified disk on a powered-down CloudProxy machine. Without necessary protection, this allows the administrator to observe application's secrets like cryptographic keys, replace program binaries with malicious programs, etc. However, we assume that the intermediate state in CPU and memory is *not* visible to the adversary during operation. This means that the adversary does not have direct access to the hardware during operation and for a few minutes thereafter (and thus cold boot attacks [13] are not possible). In practice, this assumption is reasonable because providers of Infrastructure as a Service

(IaaS) employ facilities for enclosing racks of processors in cages.

Let the protected application be a CloudProxy application whose secrets we seek to protect. CloudProxy's threat model grants the following abilities to the adversary:

1. Control of all other applications (except the protected applications) on the same machine, and programs running in other guest partitions on the TrHV. In other words, the adversary controls everything outside of the protected application's trusted computing base.

2. Physical access to all data center hardware and infrastructure, except the computer (i.e. CPU, memory, chipset, backplane, disks) that is currently running CloudProxy.

3. Control of all data center networks, and all machines that are not running CloudProxy

In this threat model, CloudProxy protects the protected application's secrets that a) reside locally on the machine, and b) are communicated to other trusted applications over an untrusted network channel. Note that CloudProxy does not defend against denial of service (DoS) or storage attack replay (for example rolling back the state of the compromised disk to an earlier state), although CloudProxy applications may implement such protections themselves. For our verification effort, we will ignore these threats (techniques to mitigate these attacks are mentioned in [20]).

2.2 Overview of CloudProxy Architecture

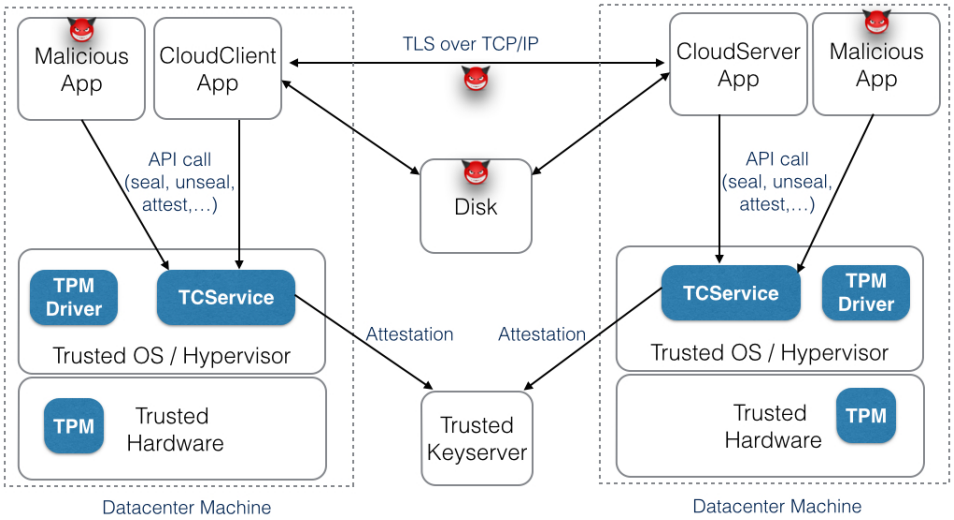

Figure 2.1: **Overview of CloudProxy architecture.** CloudProxy comprises of the trusted hardware, trusted OS / hypervisor, KeyServer and TCService. The disk and the networks are untrusted. Malicious applications may be running on the same machine as the protected applications.

Figure 2.1 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 measurement-based security principal (namely Trusted Platform Module (TPM) [21] unit) for measured boot, sealing/unsealing, and attestation. Currently, the trusted operating system (OS) is a hardened Linux kernel. At the very least, this OS protects each application's memory from being observed or modified by other applications. Chapter 3 further describes what guarantees we require from the trusted OS.

The crux of CloudProxy is the TCService process. It uses the TPM to perform cryptographic operations at initialization. TCService is the main CloudProxy component that services several mutually untrusting CloudProxy applications. It exposes a set of application programming interfaces (APIs) (see Section 2.4) for an application to a) *seal* its secrets before saving them to disk storage; b) *measure* itself and the underlying OS so as to prove that it is running unmodified code, and c) authenticate itself to other parties via the *attest* API. The applications' requests to TCService are buffered in `tcioDD`, which is a device driver running at the kernel space. `tcioDD` queues all received requests from the applications into a buffer, and dispatches the requests one at a time to TCService. This guarantees that TCService is synchronous: processing a request will not be interrupted by any subsequent requests until this current operation has finished. The return data of TCService will also go through `tcioDD` back to the caller. Hence, TCService is implemented as a single-thread process.

We briefly describe how this architecture protects us from our three threats above:

1. TCService is designed such that a malicious application cannot use the TCService API to affect a protected application's behavior. We verify this property in the present work. The OS/Hypervisor layer enforces separation from other malicious guest partitions on the same machine.

2. To protect from insider attacks that steal or modify disks, TCService provides *seal* (and *unseal*) API to add cryptographic confidentiality and integrity protection before writing secrets to disk.

3. 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, which reflects that its code and configurations are loaded as intended, the authentication protocol results in a certificate signed by KeyServer containing the application's public key.

4. Finally, CloudProxy implements a cryptographic protocol (which is a restricted version of TLS) for establishing a secure communication channel with another application.

We use an assurance case in Chapter 3 to make a systematic argument for why CloudProxy provides sufficient defense against this threat model.

2.3 Deploying and Initializing CloudProxy Applications

if is first time running **then**
    generate a new pK_A and sK_A
    $Seal_{App}(sK_A, pK_A)$
    $Seal_{TCS}(sK_{TCS}, sK_A)$
    write sealed blobs to secondary storage
else
    read sealed blobs from secondary storage
    $Unseal_{TCS}(sK_{TCS}, sealed\_sym\_key)$
    $Unseal_{App}(sK_A, sealed\_private\_key)$
end if

Figure 2.2: **Sealing / unsealing keys at initialization.** pK_A and sK_A refer to the private and symmetric key of the CloudProxy application respectively. $Seal_{TCS}(sK_{TCS}, .)$ / $Unseal_{TCS}(sK_{TCS}, .)$ refer to invoking *seal* and *unseal* API of TCService respectively, and they both use TCService symmetric key. $Seal_{App}(sK_A, .)$ / $Unseal_{App}(sK_A, .)$ refer to invoking *seal* and *unseal* API of the CloudProxy application respectively, and they both use the application's symmetric key.

For an application to use any of the CloudProxy services or security features (i.e. to run as a CloudProxy application), it has to run a CloudProxy routine for initialization. Hence, our verification assumes that CloudProxy applications correctly run initialize themselves. There are two important phases during initialization: a) CloudProxy applications get their symmetric key and private-public key pair either by generating a new set of keys, or by recovering from previously generated set of keys. b) CloudProxy applications perform remote attestation with the KeyServer.

Figure 2.2 illustrates the generation and recovering of keys during this initialization. The initialization algorithm is divided into two cases: a) the application is running for the first time or b) the application is not running for the first time. For the former case, the application will first generate a symmetric key and private-public key pair. The application will then seal the private key and symmetric key as sealed blobs, followed by writing these blobs onto secondary storage. The private key will be sealed using the application's generated symmetric key through the application's *seal* API ($Seal_{App}(pK_A)$). The generated symmetric key will be sealed using TCService symmetric key through TCService's *seal* API ($Seal_{TCS}(sK_A)$). As for the latter case, the application will read the sealed blobs from the secondary storage and unseal them accordingly.

After generation of keys, deployment of a CloudProxy application involves: a) a virtual machine image containing the trusted OS with TCService running on it, and b) the trusted KeyServer. The KeyServer is deployed with public endorsement keys of each TPM chip, de-

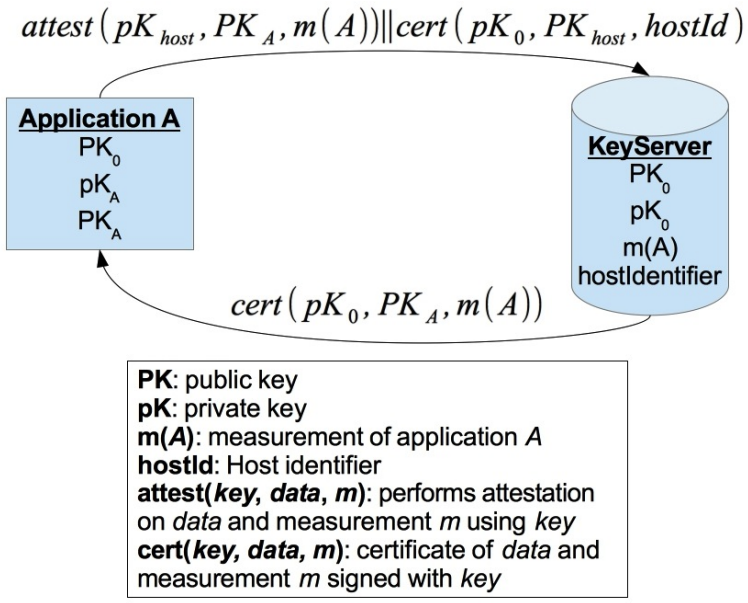

Figure 2.3: **Remote attestation of a CloudProxy application.** PK_0 and pK_0 are the public key and private key of activity owner respectively, whereas PK_A and pK_A are the public key and private key of application A respectively.

sired measurement of the host systems, TCService, and the applications. 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 the 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 2.1. To establish trust with the KeyServer, TCService, acting as part of CloudClient's host environment, measures the CloudClient application before its execution. CloudClient then 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 each application instance to its attested public key. These certificates are rooted in a public key embedded in CloudProxy components as part of program measurement. Thus, additional public key infrastructures run by third parties is not required. As a result, a program demonstrating "proof-of-possession" of a private key corresponding to the public key can authenticate itself. Since that private key is sealed by the host and cannot be revealed except to an isolated program with the same measurement, the authentication is secure from adversarial attack. Here, we are making some assumptions on the CloudProxy application: a) the application's private key is never leaked, and b) the application does not have any vulnerabilities for the attacker to exploit. CloudProxy components use the encrypted, integrity protected channel provided

by TLS to ensure the confidentiality and integrity of information exchanged between them over a public network. Figure 2.3 shows the exchanging of the asymmetric keys between an application and the KeyServer.

2.4 CloudProxy API

Once the applications have been initialized, they may invoke any of the following CloudProxy APIs, in any order. We now briefly describe the semantics of each of these APIs (formal semantics in [20]).

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 part of the signed information in this 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, and then attaches the message authentication code (MAC) of 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.
7. *StartApp(filename)*: Fork a new application process.

Chapter 3

Assurance Case For CloudProxy

Our primary goal is to prove that CloudProxy protects its client applications from the threats allowed in our threat model. These security guarantees are quite informal, and hence do not translate to statements in a formal language. Our first contribution in this work is to formalize these high-level security properties into a set of axioms, assumptions, and lemmas that are provable on a CloudProxy model. Although we formalize our assumptions and lemmas, we use an informal assurance case as a meta-level argument for why our lemmas and assumptions fulfill the high-level security properties. In Chapter 4, we construct a formal model of CloudProxy, and in Chapter 5, we prove a subset of our lemmas on this model. This model will act as a golden specification for all future revisions to CloudProxy's implementation.

An assurance case is a documented body of evidence that provides a systematic, albeit informal, argument that a system satisfies a set of properties [3]. 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 [22, 25]. There have been a few adoptions of assurance cases for safety critical systems in literature and in the industry [27, 15]. We believe that through the employment of assurance case framework, we will be able to make the argument for our proofs clearer, more systematic and consistent. Domain experts would also have a common documentation for checking and analyzing the entire verification process. However, note that a limitation to this work is that this assurance case has yet to be examined by multiple domain experts.

3.1 Goal Structuring Notation (GSN)

There are a few notations for expressing assurance cases. We follow the goal structuring notation (GSN) [16, 11] for our assurance case framework described in [22]. For the rest of this section, we will only use a subset of GSN, and the following defines this subset of elements [11]:

- *Goal*: A claim forming part of the argument.

- *Strategy*: A description of the nature of the inference that exists between a goal and its supporting goal(s).
- *Evidence*: A supporting proof for a claim.
- *Context*: Information describing the context of the referenced element.
- *Assumption*: An intentionally unsubstantiated statement.

GSN also defines the representation of the relationships between these elements:

- *SupportedBy*: An inferential (an inference between goals in the argument) or evidential (the link between a goal and the evidence used to substantiate it) relationship. Permitted connections are: goal-to-goal, goal-to-strategy, goal-to-solution, strategy-to-goal.
- *InContextOf*: A contextual relationship. Permitted connections are: goal-to-context, goal-to-assumption, goal-to-justification, strategy-to-context, strategy-to-assumption and strategy-to- justification.

Figure 3.1 shows the graphical representation of the elements and Figure 3.2 shows the graphical representation of the elements relationships.

For evidence elements that have dotted outline, we have not completed the proofs for these evidences and are either work in progress or future works.

3.2 Structuring CloudProxy Assurance Case in GSN

In this section, we present our argument from a top-down approach by iteratively decomposing into sub-claims and finally proving each sub-claim with evidence or through assumptions. Note that our argument for the decomposition into sub-claims is informal and may be incomplete, and thus there may be possible modifications in the future.

CloudProxy is too complex to verify in its entirety. However, it can be modularized into different components. As shown in Figure 2.1, CloudProxy relies on several components: a trusted hardware, a trusted OS/Hypervisor layer, to-be-verified TCService, and a trusted remote key server. In figure 2.1, we separate CloudProxy into a trusted hardware, a trusted OS/Hypervisor layer, TCService, and applications. In this work, we only verify TCService, and assume that properties about other components hold. This is encoded as assumption **A1** in Figure 3.3: the hardware, the Hypervisor, and the OS (including the TPM driver) are trusted. We are aware of orthogonal efforts [12] on verifying security properties of hypervisors, TLS protocol implementation, etc.

Proving that CloudProxy protects the protected application's secrets (**G1**) is decomposed 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. Consequently, verifying application logic is out of scope

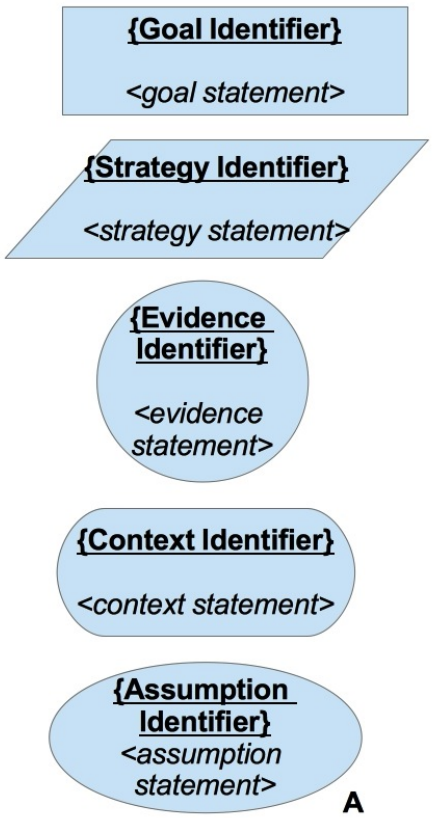

Figure 3.1: **Elements in GSN.** Elements in GSN[11].

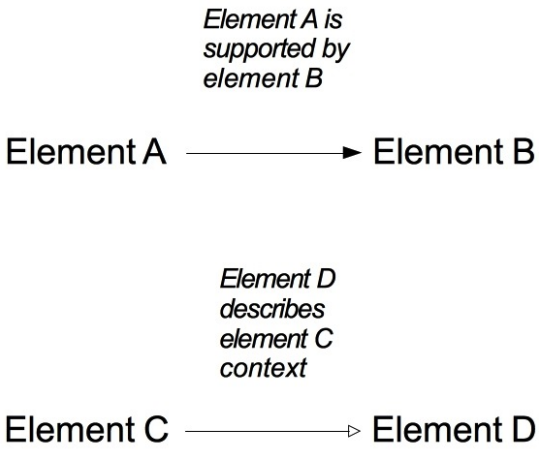

Figure 3.2: **Relationships among elements in GSN.** Relationships among elements in GSN[11].

(**Ct1**). Each goal in **G2** - **G4** is realized by 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 verification of the measured launch sequence (**E1**), which uses the TPM to compute a cryptographic hash of the binaries before launching TCService and applications. The memory protection of OS/Hypervisor layer (**A1**) obviates the need for measuring binaries after launch. In addition, based on our threat model assumption, an insider is not able to access the memory chip of the machine that is running CloudProxy (**A2**). All top-level security goals **G2** - **G4** depend on **G7** because successfully mounting a modified TCService binary will nullify all security guarantees. **G5** and **G6** together guarantee that a protected application's secret is never revealed in plaintext to an adversary on the same machine as the protected application. **G5** enforces that a malicious program does not observe a protected application's execution. Our notion of execution only considers an application's state updates; we do not consider information leaks via side channels, or via channels intended for communication between applications (e.g. network). **G6** enforces that the protected application's secrets have cryptographic confidentiality and

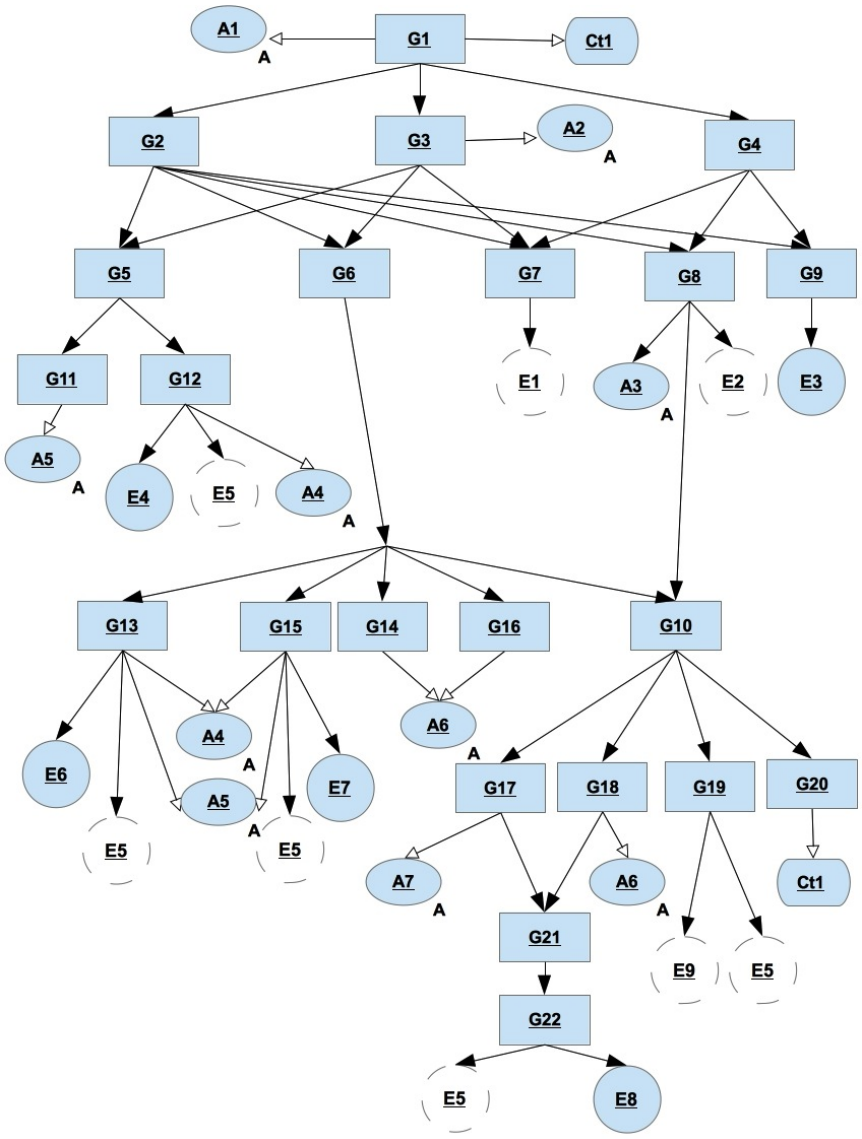

Figure 3.3: **CloudProxy assurance case.** The CloudProxy assurance case shows the top-level goal, **G1**, is iteratively decomposed into subgoals (square nodes), evidences (circle nodes) and assumptions (oval nodes). **Ct1** defines the context for goal **G1** and **G20**. Table 3.1 describes each node in detail.

integrity protections before being saved to disk. **G8** and **G9** together protect an application's secret that is sent over the network. **G8** is needed for authenticating mutually trusting applications (**E2**) over an untrusted network. Consider Figure 2.1 where CloudServer must authenticate a request from CloudClient. TCService attests to CloudClient's measurement, which allows CloudServer to verify CloudClient's identity. Following remote attestation, **G9** enforces that future communication takes place over a cryptographically secure channel.

CloudProxy uses a restricted version TLS [20] (**E3**) for secure communication. We do not verify this TLS implementation in this work.

Consider the assurance case for **G5**: no malicious application can compromise TCService or the protected applications. This responsibility is shared between the OS protections (**G11**) and the TCService API guarantees (**G12**). **G11** stipulates that our OS a) protects an application's address space from reads or writes by other programs, and b) protects the TPM driver's address space from other malicious programs. Both requirements can be fulfilled by a separation kernel [23]. While separability is a strict requirement (and possibly unreasonable for commodity OS), we assume for this discussion via **A4** that we have a separation kernel. With OS-enforced separation between protected applications and malicious applications, the 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 are independent of the malicious applications'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 behaviours in the implementation are captured by the model. However, model validation is still a work in progress (see Chapter 6).

Consider the assurance case for **G6**: protected application's secrets have cryptographic confidentiality and integrity protections before being written to disk. These secrets must be sealed using TCService's *seal* API. With this guarantee, 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: a) **G13-G16**: TCService's implementation of *seal* preserves confidentiality and integrity, and b) **G10**: TCService never reveals its sealing key. We make a crucial assumption (**A6**) that we have a Dolev-Yao adversary [14]. Analyzing the strength of cryptographic operations is beyond our scope. In other words, our proof assumes axioms of strong encryption, pre-image resistance of hash functions, 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 strong collision resistance axiom about hash function used in MAC.

TCService also appends the application's measurement within the sealed secret. The measurement is used to decide if it should *unseal* a sealed secret on behalf of an application. An application's measurement must match the measurement that is sealed together with the secret. Therefore, we also need goals **G13** (fulfilled by **E6**) and **G15** (fulfilled by **E7**) to prove that TCService does not incorrectly *unseal* the protected application's secret on behalf of the malicious application. We further assume in **A4** that our OS / hypervisor layer enforces separation between all applications and TCService, or else the malicious application can exploit the OS to observe secrets. While building a formal model, we uncovered an undocumented assumption **A5** that the OS does not reuse process identifiers at any point of time — the process identifier (*pid*) is used to identify the application invoking the API call. In other words, once the OS has generated a *pid* for an application, even after this

application has terminated, there will never be any other subsequent application that has this same *pid*.

Consider the assurance case for **G10**: the protected application and TCService do not reveal keys needed for attestation and sealing. We must prove that this property holds during a) TCService's initialization (**G17**), b) application's initialization (**G18**), and c) servicing of API request by TCService (**G19**). Note that verifying application logic is out of scope, but the CloudProxy application's initialization is handled by CloudProxy. This initialization is verified by **G18**. 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**. Since the TPM driver and the crypto-library are trusted (**A6** and **A5**), we use their axioms to verify the remainder of the initialization routine (**G21**).

E8 fulfills **G22** by proving that that each write (e.g. file write, socket send) out of the process sandbox 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 though we prove non-interference in **G12**. This is because TCService may leak the protected application's secrets by erroneously revealing its own sealing key.

Table 3.1: **Descriptions of assurance case nodes. Node** refers to the the assurance case node in Figure 3.3. **Proof Obligations** are either nodes in the assurance case, or property number(s) in Chapter 5.

Node	Description	Proof Obligation
A1	Hardware, Hypervisor and OS are trusted.	
A2	Adversary cannot physically access computer currently running CloudProxy.	
A3	KeyServer is trusted.	
A4	Hypervisor and OS layers enforce separability.	
A5	OS will not reuse PID.	
A6	Perfect cryptographic primitives.	
A7	TPM driver does not leak TCService's secrets.	
Ct1	Verifying app logic (excluding CloudProxy initialization) is out of scope.	
E1	Verify measured launch mechanism.	
E2	Verify remote attestation protocol.	
E3	Use restricted version of TLS for network communication.	
E4	Prove **G12** on UCLID model.	Ppty (5.3)-(5.4), (5.8)-(5.9)
E5	Validate UCLID model.	
E6	Prove **G13** on UCLID model.	Ppty (5.13)
E7	Prove **G15** on UCLID model.	Ppty (5.14)
E8	Prove **G22** on UCLID model.	Ppty (5.16)
E9	Prove **G19** on UCLID model.	
G1	CloudProxy 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.	**A2**, **G5-G7**
G4	Secure against network attacks.	**G7-G9**
G5	No malicious app can compromise TCService or protected app.	**G11-G12**
G6	Data confidentiality and integrity of protected app's secrets.	**G10**,**G13-G16**
G7	Protected app and TCService should be launched from unmodified code.	**E1**
G8	Remote attestation through untrusted channels.	**A3**, **E2**, **G10**
G9	Use cryptographic protocol for app's communications.	**E3**
G10	Protected app and TCService do not reveal attestation and sealing keys.	**G17-G20**
G11	Isolation of apps memory space from other apps.	**A5**
G12	Non-interference of protected apps through TCService APIs.	**A4**, **E4-E5**
G13	TCService Seal API provides data confidentiality.	**A4-A5**, **E5-E6**
G14	Cryptographic Seal provides data confidentiality.	**A6**
G15	TCService Seal API provides data integrity.	**A4-A5**, **E5**, **E7**
G16	Cryptographic 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 algorithm does not reveal keys.	**G22**
G22	Plaintext-writes do not leak keys.	**E5**, **E8**

Chapter 4

CloudProxy Abstraction

Formal verification is a resource-intensive process. Therefore, it is often infeasible to perform formal verification for the entire system. Our assurance case in Chapter 3 allows us to focus our verification effort on the composition of TCService with the protected and malicious applications. We have assumed that the OS, hypervisor and hardware are trusted, and hence we need not precisely model the entire trusted computing base. In other words, we are only focusing our verification effort mainly on TCService and part of CloudProxy framework which CloudProxy applications use.

In this chapter, we will describe our formal model for CloudProxy which captures the components and behaviors that we wish to verify. Verifying properties on this model will be discussed in Chapter 5. Besides proving properties, this model may also serve as a golden specification for all future revisions to CloudProxy's implementation.

4.1 Modeling in UCLID

Figure 4.1 presents the structural overview of our model[1], for which we use the UCLID [6] modeling language.

This model is a synchronous composition of four transition systems:

1. *App* (protected application);
2. *Mal_App* (malicious application);
3. *Scheduler*, and
4. TCService.

Our model assumes one protected application and one malicious application. We are making this simplification of modeling one malicious application instead of arbitrary number of malicious applications because the properties in Chapter 5 are reasoning over one malicious

[1]The model is available at the URL: `http://uclid.eecs.berkeley.edu/cloudproxy`

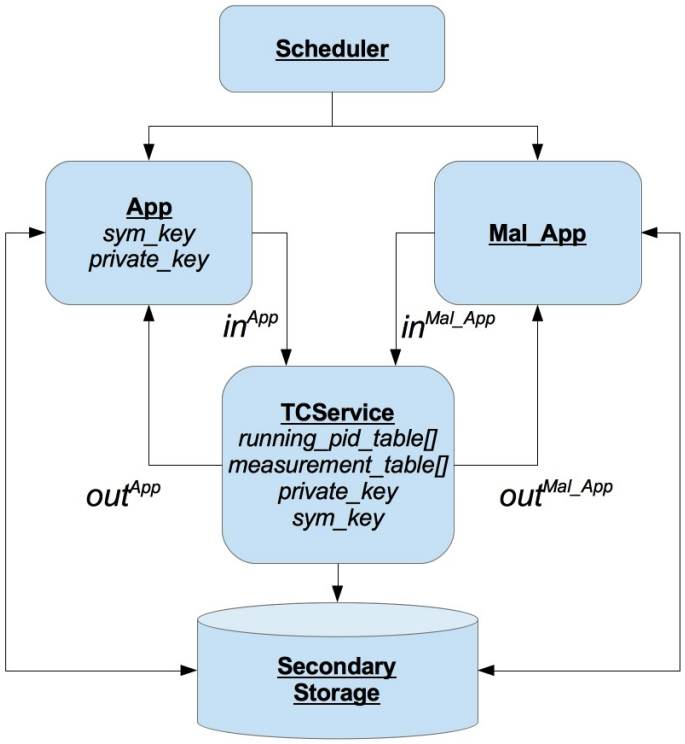

Figure 4.1: **CloudProxy model in UCLID.** The CloudProxy UCLID model is a synchronous composition of the transition systems *App*, *Mal_App*, *Scheduler*, and TCService. An arrow shows the data flow between the transition systems. *Secondary_Storage* is a set of state variables which other components can read values from or write values to.

application only. Our model captures the initialization routine 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. For example, a file server may erroneously respond to a malicious application's request with the protected application's file. Therefore, we verify TCService in the presence of an arbitrary *App* and an arbitrary *Mal_App*. Note that since we are verifying CloudProxy, verification of a specific application's logic is beyond scope.

In the CloudProxy implementation, the applications may invoke TCService API calls non-deterministically and asynchronously. We model this behaviour by having the *Scheduler* non-deterministically trigger either *App* or *Mal_App* to execute in each step. When triggered, *App* and *Mal_App* non-deterministically choose an API call and arguments to TCService in each step of execution.

One difference between *App* and *Mal_App* are that *App* has a symmetric key (*sym_key*) and a private key (*private_key*), which are modeled as state variables. These state variables

are used for modeling CloudProxy application initialization. *Mal_App* does not need to have the mentioned state variables, since it may run without using the CloudProxy initialization code. Thus, the initialization logic, as described in Section 2.3, is only implemented in *App*. Another difference is that we fix a symbolic term to represent *App*'s process ID (*pid*), and any other *pid* that is not equal to *App pid* will be considered as *Mal_App*'s *pid*.

In Section 2.2, we discussed how TCService uses a device driver called `tcioDD` to buffer all API requests, and handles each request synchronously. Thus, we model TCService as a sequential system, treating computation for each API to be an atomic state update. Here, we assume that `tcioDD` has infinite buffer. As a result, we assume no loss in requests due to filled buffer. We implement the semantics of each API from Section 2.4.

TCService maintains the following state variables: a) a private key (*private_key*) for remote attestation, b) a symmetric key (*sym_key*) for use in *seal* and *unseal*, c) *running_pid_table[]* for process identifiers of all running applications, and d) measurements *measurement_table[]* of all running applications. Each API may involve reading and writing to *Secondary_Storage*, which is modeled as an unbounded memory. In the TCService implementation, TCService has a linked-list to keep track of the running applications and their measurements (i.e. the hash of the application binary file). Each node in the linked-list contains the *pid* and its measurements. We abstract this into an unbounded array in the theory of Arrays, where the array maps integers to integers. Both *running_pid_table[]* and *measurement_table[]* are unbounded array data types in our model. The former maps the *pid* to boolean `true` or `false`, whereby `true` implies the process with that *pid* is running, and false otherwise. The latter maps the *pid* to the measurements.

4.2 Capabilities of *Mal_App*

The following summarizes the assumptions on the capabilities of the malicious applications (*Mal_App*) in our model:

1. *Mal_App* is able to execute any cryptographic functions as well as invoke any API of TCService.

2. *Mal_App*, just like *App*, can be started by TCService.

3. At initial state, *Mal_App* does not have the knowledge of either *App* secrets or TCService keys in plaintext.

4. *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. This also implies that `tcioDD`, the buffer that sends and receives data between TCService and the applications, is protected from eavesdropping.

5. 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.

4.3 Model Assumptions and Axioms

In our UCLID model, we use uninterpreted functions and terms to abstract functions and variables in the C++ implementation code respectively. To make our uninterpreted functions meaningful, we impose some restrictions on the behaviors of these functions through axioms.

Let $\mathbb{M}$ be the set of measurements, $\mathbb{P}$ be the set of *pid*, $\mathbb{D}$ be the set of data. K_{TCS} is the symmetric key of *TCService*, PID_{App} is the *App pid*, and PID_{Mal_App} is the *Mal_App pid*. We also define M_{App} to be the measurement of *App* and M_{Mal_App} to be the measurement of *Mal_App*. The following is the list of axioms implemented in the model:

1. Authenticated encryption and decryption (*ENC_MAC()*, *DEC_MAC()*):

$$\forall x \in \mathbb{M} : DEC\_MAC(ENC\_MAC(x, K_{TCS}), K_{TCS}) = x$$
$$\forall x, y \in \mathbb{M} : (DEC\_MAC(x, K_{TCS}) \neq DEC\_MAC(y, K_{TCS}) \Leftrightarrow (x \neq y)$$

2. Concatenation and extraction of terms (*EXTRACT_1ST()*, *EXTRACT_2ND()*, *CAT_2_PARAMS()*):

$$\begin{aligned}\forall x \in \mathbb{D} : &(EXTRACT\_1ST(CAT\_2\_PARAMS(x, M_{App})) = x) \wedge \\ &(EXTRACT\_1ST(CAT\_2\_PARAMS(x, M_{Mal\_App})) = x) \\ \forall x \in \mathbb{D} : &(EXTRACT\_2ND(CAT\_2\_PARAMS(x, M_{App})) = M_{App}) \wedge \\ &(EXTRACT\_2ND(CAT\_2\_PARAMS(x, M_{Mal\_App})) = M_{Mal\_App})\end{aligned}$$

3. Cryptographic hash function (*SHA256()*):

$$\forall x, y \in \mathbb{D} : (x \neq y) \Leftrightarrow (SHA256(x) \neq SHA256(y))$$

We also have a list of assumptions for our model:

1. The *pid* of *App* and *Mal_App* are not the same:

$$M_{App} \neq M_{Mal\_App}$$

2. Let $MAL\_APP\_BIN\_FILES\_SET$ be a predicate which returns true if the argument belongs to the set of *Mal_App* binary files, and false otherwise. We also define GET_BIN_FILE as a function that takes in a file name and returns the binary of the file. *Mal_App* should not have the same binary as *App*:

$$\neg MAL\_APP\_BIN\_FILES\_SET(GET\_BIN\_FILE(APP\_FILE\_NAME))$$

3. *Mal_App* should not know *App* secret initially.

We derive these three stated assumptions from both our domain expertise as well as from our properties verification (see Chapter 5). The first assumption (that the *pid* of *App* and *Mal_App* are not the same) is an important one. In other words, we are assuming that the trusted operating system will not reuse *pid* for new processes. Chapter 5 discusses how this assumption affects the properties we are verifying.

For the second assumption, the premise enforced by the CloudProxy host is that *TCService* executes the very same binary as intended, and hence the binaries of *Mal_App* would not be the same as the *App* binary program. In other words, we are assuming that time-of-check-to-time-of-use (TOCTTOU) attack is not possible. This is a result from CloudProxy threat model, which ensures that the physical memory can only be modified by adversary when there is no CloudProxy program running on it. Even if the adversary arranges *Mal_App* to have the same binary as *App*, *Mal_App* would behave the same as *App*. This implies the following two assumptions: a) *App*, when running a single copy of itself on its own, is secure, and b) *App*, composed of multiple copies of itself, is secure.

The last assumption is that the *Mal_App* should not know *App* secrets initially. Clearly, this must hold for the verification effort to be meaningful.

One challenge is that the instantiation of these axioms may cause a memory blowup during the verification phase. Thus, we only list the axioms necessary for our properties in the subsequent chapters. Also, whenever possible, we manually instantiate terms (such as the *Mal_App pid*) instead of reasoning on a universal quantifier over the entire domain to circumvent this memory blowup problem.

Chapter 5

Verification

In this chapter, we formalize and verify properties of the UCLID model for each evidence in our assurance case as presented in Chapter 3. Subsequent sections will describe the details of each property, as well as the approach in verifying them using the UCLID decision procedure. The evidences marked with a dashed line represent proofs that are currently in progress or left for future work.

5.1 Property 1: Non-interference

G11 in Figure 3.3 stipulates that TCService exhibits non-interference: the responses to an application's API requests are independent of the malicious applications's API requests. Consider a system that has inputs and outputs from two users: *App* and *Mal_App*. Both *App* and *Mal_App* are treated as the *environment* inputs of *TCService*, where they can non-deterministically choose any API request and arguments. Informally, the non-interference property states that *Mal_App*'s inputs can be removed without affecting *App*'s outputs, and vice versa. In the context of CloudProxy, non-interference requires two checks:

1. **secure information flow**: *App*'s secrets are not leaked to *Mal_App* when *Mal_App* invokes an API request.

2. **non-interference**: the 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 **secure information flow** to

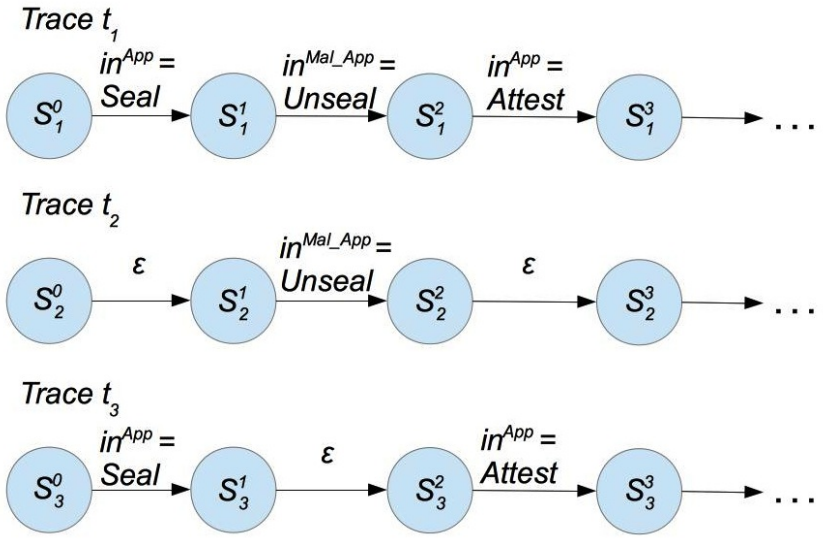

Figure 5.1: **Non-interference property for CloudProxy.** The figure shows three traces $t1$, $t2$ and $t3$, where trace $t2$ replaces *App* API requests in $t1$ with ε, and $t3$ replaces *Mal_App* API requests in $t1$ with ε.

Mal_App:

$$\forall t_1, t_2 \in T : (in^{App}(t_2) = \varepsilon \wedge in^{Mal\_App}(t_1) = in^{Mal\_App}(t_2)) \Rightarrow (out^{Mal\_App}(t_1) = out^{Mal\_App}(t_2)) \tag{5.1}$$

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

$$\forall t_1, t_3 \in T : (in^{Mal\_App}(t_3) = \varepsilon \wedge in^{App}(t_1) = in^{App}(t_3)) \Rightarrow (out^{App}(t_1) = out^{App}(t_3)) \tag{5.2}$$

where ε denotes no API invocation (modeled as stuttering steps). Note that this definition only applies if the following two conditions are met:

1. TCService must be deterministic (*App* and *Mal_App* need not be deterministic), and
2. TCService must be total with respect to inputs.

A hyperproperty is a set of sets of infinite traces [8]. As properties (5.1) and (5.2) reason over a pair of sets of traces, they are both are hyperproperties. We can rewrite them as 2-safety properties [8] and prove them using induction.

As Figure 5.2(a) illustrates, we construct a 2-fold parallel self-composition of the system, resulting in a pair of traces t_1 and t_2. In our presentation, we close the system (TCService) together with its environment (*App* and *Mal_App*), treating inputs and outputs as functions of system state. Let $\mathcal{I}$ be the set of all inputs to the system, and $\mathcal{O}$ be the set of all outputs

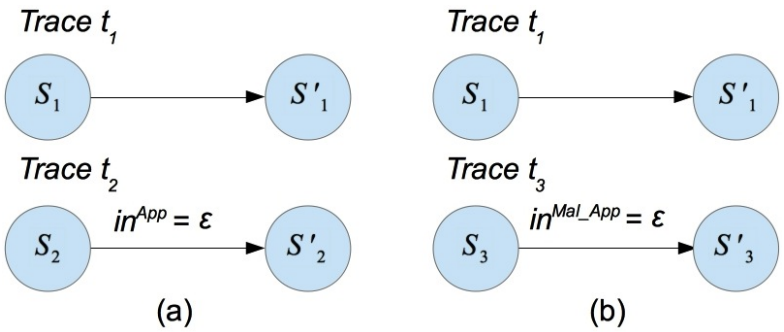

Figure 5.2: **Proving non-interference in UCLID.** S denotes the state of TCService in our UCLID model. We prove secure information flow in (a) by proving that Mal_App cannot distinguish s'_1 from s'_2. We prove non-interference in (b) by proving that App cannot distinguish s'_1 from s'_3.

from the system. $R \subseteq S \times in \times S$ is the transition relation of TCService over set of states S, where $in \in \mathcal{I}$. For TCService's state s, we use in^{App} and in^{Mal_App} to refer to App's input to TCService and Mal_App's input to TCService respectively, where $in^{App}, in^{Mal\_App} \in \mathcal{I}$. $out^{App}(s)$ and $out^{Mal\_App}(s)$ refer to TCService's output to App at state s and TCService's output to Mal_App at state s respectively, where $out^{App}(s), out^{Mal\_App}(s) \in \mathcal{O}$. Let Sys_1 and Sys_2 be the two instances of the system, which we let them run in parallel. Let s_1 be the state in Sys_1, and s_2 be the state in Sys_2. We define R_1 to be the transition relation of Sys_1, and R_2 to be the transition relation of Sys_2. We also define in_1 be the input of Sys_1, and in_2 be the input of Sys_2. For **secure information flow**, we prove the following inductive property:

$$\forall s_1, s_2. Init(s_1) \wedge Init(s_2) \Rightarrow \Phi_{Mal\_App}(s_1, s_2) \tag{5.3}$$

$$\begin{aligned}&\forall s_1, s'_1, s_2, s'_2, in.\\ &(\Phi_{Mal\_App}(s_1, s_2) \wedge R_1(s_1, in, s'_1) \wedge R_2(s_2, in, s'_2)) \Rightarrow\\ &\Phi_{Mal\_App}(s'_1, s'_2)\end{aligned} \tag{5.4}$$

where

$$\begin{aligned}&\Phi_{Mal\_App}(s_a, s_b) \doteq\\ &\forall s'_a, s'_b. R(s_a, in, s'_a) \wedge R(s_b, in, s'_b) \Rightarrow\\ &(out^{Mal\_App}(s'_a) = out^{Mal\_App}(s'_b))\end{aligned} \tag{5.5}$$

$$R_1(s, in, s') = R(s, in, s') \tag{5.6}$$

$$R_2(s, in, s') = (R(s, in, s') \wedge in^{App} = \varepsilon) \tag{5.7}$$

Note that s_1, s_2, s_1', s_2' represent internal states of TCService. 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. Since output variables are part of state, we enforce indistinguishability of two states by invoking the same, albeit arbitrary, API request from both states and comparing the output variables in the next states. Property 5.3 checks the base case that Φ holds on any pair of initial states. This is a trivial proof because TCService is always initialized to a concrete initial state. The inductive step (property 5.4) 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*. Due to insufficient quantifier instantiation, we needed an auxiliary inductive invariant Ψ_{aux}: the component of TCService state that affects *Mal_App* is identical in s and t.

Proving **non-interference** between *App* and *App* requires a similar inductive proof. For conciseness, we only list the property here; the above discussion applies verbatim if *App* is substituted for *Mal_App* for each other. *Mal_App*'s API requests does not affect *App*'s API observations if:

$$\forall s_1, s_3. Init(s_1) \wedge Init(s_3) \Rightarrow \Phi_{App}(s_1, s_3) \tag{5.8}$$

$$\begin{aligned} &\forall s_1, s_1', s_3, s_3', in. \\ &\Phi_{App}(s_1, s_3) \wedge R_1(s_1, in, s_1') \wedge R_2(s_3, in, s_3') \Rightarrow \\ &\Phi_{App}(s_1', s_3') \end{aligned} \tag{5.9}$$

where

$$\begin{aligned} &\Phi_{App}(s_a, s_b) \doteq \\ &\forall s_a', s_b'. R(s_a, in, s_a') \wedge R(s_b, in, s_b') \Rightarrow \\ &(out^{App}(s_a') = out^{App}(s_b')) \end{aligned} \tag{5.10}$$

$$R_1(s, in, s') = R(s, in, s') \tag{5.11}$$

$$R_2(s, in, s') = (R(s, in, s') \wedge in^{Mal\_App} = \varepsilon) \tag{5.12}$$

UCLID took about 40 seconds to prove each property. [1]

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

5.2 Property 2: Data Confidentiality

In this section, we describe our proof of **G6**: *Mal_App* cannot acquire the plaintext of a sealed secret belonging to *App*. Recall from Figure 3.3 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 3.3). We assume a Dolev-Yao [9] model where such attack is infeasible.
- **Lemma 2**: *Mal_App* cannot obtain the plaintext by invoking a sequence of CloudProxy API calls (goal **G13** in Figure 3.3).

Lemma 1 is simply assumed in our work since we assume a Dolev-Yao adversary. In accordance with the Dolev-Yao model [9], our model represents data as terms of some abstract algebra, and cryptographic primitives operate on those terms to produce new terms. We have also used ProVerif [2], an automatic cryptographic protocol verifier which assumes a Dolev-Yao adversary, to trivally prove this lemma.

Proving **Lemma 2** is necessary because TCService implements the following logic (by appending measurement to the secret prior to sealing) for deciding if it should fulfill an *unseal* API request: *After unsealing, if the secret's measurement does not match the measurement of API caller, then the request fails.*

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 a cryptographic seal function that first encrypts the plaintext, and then appends an integrity-protecting MAC of the plaintext. Let $\mathcal{I}$ be the set of all inputs to the system. $R \subseteq S \times in \times S$ is the transition relation of TCService over set of states S, where $in \in \mathcal{I}$. 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. $out_{result}^{Mal\_App}(s)$ is the return output of TCService to the *Mal_App* which has invoked the TCService API. Finally, sK_{TCS} denotes the symmetric key used by TCService to *seal* or *unseal*. We define **Lemma 2** as follows and prove it via 1-step induction:

$$\begin{aligned}\phi(s) \doteq & \forall secret \in \mathbb{D}, s'. \\ & (in_{API}^{Mal\_App} = unseal \wedge R(s, in^{Mal\_App}, s') \wedge \\ & in_{arg}^{Mal\_App} = ENC\_MAC(sK_{TCS}, secret, m_{App})) \Rightarrow \\ & out_{result}^{Mal\_App}(s') \neq secret \end{aligned} \tag{5.13}$$

where $ENC\_MAC(sK_{TCS}, secret, m_{App})$ is a term encoding any sealed secret that can belong to *App*, if *secret* is an unconstrained symbolic constant. This allows us to only consider API calls whose argument has this form.

As a result, property (5.13) 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.

One possible scenario where *Mal_App* may learn the secret of *App* without violating (5.13) is when *Mal_App* is able to *somehow* learn $ENC\_MAC(sK_{TCS}, secret, m_{Mal\_App}$. We argue that this is not possible without *Mal_App* knowing either the secret beforehand. As we have assumed that the authenticated encryption function $ENC_MAC()$ is unbreakable (**A6**), this means that there is no way of generating $ENC_MAC(key, x, y)$ unless all three key, x and y are known.

UCLID took about 30 seconds to prove this property. Moreover, we discovered the following necessary assumptions to prevent spurious counter-examples to the inductive proof:

1. *Mal_App* has a different measurement than *App*, i.e. $m_{Mal\_App} \neq m_{App}$, and
2. OS must guarantee that every process has a unique *pid* throughout at all times.

For the first assumption, besides assuming the usage of collision-free hash functions, we need to argue that neither the path name of the malicious application is identical to the benign application one nor are their binaries identical. One possible violation to this assumption is the time of check to time of use (TOCTTOU) vulnerability. Malicious application A may first request TCService to start application B via *startapp*. When TCService reads in B's path name and about to execute the binary, A swaps the binary of B to another malicious application binary C. This results in TCService starting C instead of B. However, this is not possible under the threat model of CloudProxy (see Section 2.1), since the adversary is only allowed to access the disk and swap its content when CloudProxy is not running any program on this disk.

The second assumption is necessary for this confidentiality to hold. If *pid* can be reused, a malicious application is able to masquerade itself as the protected application by having the same *pid* as this protected application. By invoking *unseal* on any sealed data of the protected application, TCService will be tricked by the fake *pid* and the matching measurements, and thus unseal the data for malicious application.

5.3 Property 3: Data Integrity

Besides data confidentiality, we also need to prove that there is no way *Mal_App* may make *App* unseal data tampered by *Mal_App* itself for **G6**. Again, we assume perfect integrity protection of the cryptographic seal function $ENC\_MAC(key, ., .)$, and hence any modification to $ENC\_MAC(key, ., .)$ should *not* be able to *unseal* successfully (**G16**). We have used ProVerif [2] to trivally show *authenticity* for this lemma. Only data that was previously sealed by TCService can be successfully unsealed by TCService (**G15**). 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 in_{API}^{App} be the API call from the *App* to TCService, and let in_{arg}^{App} be the arguments of the API call from the *App* to TCService. $out_{success}^{App}(s)$ is the return status of TCService to the *App* which has invoked the TCService API. The status takes a boolean value, and shows whether the invocation of the API is sucessful. Let $\mathbb{M}$ be the set of measurements, and $\mathbb{D}$ be the set of data. We prove that an *unseal* request satisfies:

$$\begin{aligned}\phi(s) \doteq & \forall secret \in \mathbb{D}, \forall m \in \mathbb{M}, s'. \\ & (in_{API}^{App} = unseal \wedge R(s, in^{App}, s') \wedge \\ & in_{arg}^{App} = ENC\_MAC(sK_{TCS}, secret, m) \wedge m \neq m_{App}) \Rightarrow \\ & \neg out_{success}^{App}(s')\end{aligned} \tag{5.14}$$

where $ENC\_MAC(sK_{TCS}, secret, m)$ is a term encoding any sealed secret that can belong to any application other than *App*, if *secret* and m are unconstrained symbolic constants. This allows us to only consider API calls whose argument has this form.

UCLID took less than 5 seconds to prove this property. Again, we require the same assumptions to prevent spurious counter-examples to the inductive proof:

1. *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.

2. OS must guarantee that every process has a unique *pid* throughout at all times.

In Section 5.2 we have discussed the first assumption. The following illustrates an attack without the second assumption. If a malicious application masquerades itself as the protected application by having the same *pid* as this protected application, it can request TCService to seal some malicious data. This piece of malicious data will be sealed together with the same measurement as the protected application. Hence, the protected application will be able to unseal this sealed malicious data through TCService without an error.

A caveat to note here is that CloudProxy does not 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 Property 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}. This is because both TCService and the CloudProxy applications use similar piece of initialization code, where the only difference is the functions used for cryptographic operations. TCService uses the functions provided by the TPM driver for cryptographic operations such as authenticated encryption, whereas *App* uses the crypto-library provided by CloudProxy. In this section, we prove that keys

sK_{App} and pK_{App} are never leaked in values written to disk (goal **G18**). We only focus our attention on *App*'s keys in this section. We argue that the property and proof for TCService is identical due to the same initialization code. However, we will need to further extend our model to capture the initialization phase of TCService, as well as to abstract the TPM to prove this explicitly. We express this property in the semantic information flow framework introduced by [19]. 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 outputs along the two traces must be identical. 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.

$$\mathcal{S}(x) = \begin{cases} true & in_{API}^{App} = seal \wedge x = ENC\_MAC(sK_{TCS}, in_{arg}^{App}, m_{App}) \\ true & init^{App} = true \wedge x = ENC\_MAC(sK_{App}, pK_{App}, m_{App}) \\ old(S(x)) & \text{otherwise} \end{cases} \tag{5.15}$$

where $init^{App}$ is a flag that determines whether *App* is at the initalization 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:

$$\begin{aligned} \forall s_1, s_2, s_1', s_2'. \\ & (s_1 \setminus \{pK_{App,1}, sK_{App,1}\}) = (s_2 \setminus \{pK_{App,2}, sK_{App,2}\}) \\ & \wedge R(s_1, in^{App}, s_1') \wedge R(s_2, in^{App}, s_2') \\ & \wedge (\neg\mathcal{S}(out^{disk,1}(s_1')) \vee \neg\mathcal{S}(out^{disk,2}(s_2'))) \Rightarrow \\ & (out^{disk}(s_1') = out^{disk}(s_2')) \end{aligned} \tag{5.16}$$

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. The soundness of this proof relies on the model validation proof; model validation would prove that we have captured all possible file writes in our model.

Chapter 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 have made first steps in using KLEE [7] to validate our UCLID model against the C++ implementation, using techniques in [26].

Since we do not precisely model all computation within TCService (crypto libraries are abstracted away via axioms), we need to argue that the unmodeled code does not affect the subset of TCService state that we do model. Let $\mathcal{V}$ denote the state variables that are present in our UCLID model. Then, we manually identify code paths that will be *pruned* away from our modeling. Finally, 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 [26]. 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 [26]. Both validation steps are a work in progess.

The entire CloudProxy has about 58k lines of code (LoC), and the code that we are working on is only about 8k compilable LoC (mainly containing TCService implementation and initialization code). The cryptographic keys, measurement table, and the *pid* table in TCService are our $\mathcal{V}$ set, and only approximately 1k LoC modifies $\mathcal{V}$. We perform DMV by creating a shadow variable for each variable in $\mathcal{V}$, and then assert the values of the state variables and their corresponding shadow variable are the same right after the pruned code fragments. As noted in [26], while this check only ensures that the relevant variables are modified at the boundaries of the pruned code, it is still useful in finding modeling errors. To run KLEE, we need to change the `main` function of TCService code to run a bounded number of loops which services CloudProxy applications requests. These requests are made symbolic. Other code that we have also made symbolic are the cryptoprimitives outputs and some of the system calls (such as `read()`). After this initial effort in DMV, we model in UCLID the remaining 1K LoC. Completing DMV is still a work in progress. We need to overcome the scalability issue as our DMV approach is still mostly manual, and KLEE takes a very long time (more than an hour) to run just one iteration of the loop which generates

TCService requests.

We encountered several challenges in performing OMV, and delay that to future work. Some challenges include:

1. Cryptographic primitives validation;
2. Term-level abstraction validation (as compared to using bit-vectors abstraction), and
3. Show that the usage of the uninterpreted functions and predicates are a sound abstraction of the data structures (for example linked-lists) used in the code.

In general, the validation approach presented in [26] may still be applicable, except with some minor changes. For instance, we need to convert the tuple of a pointer to some data and the length of this data into a term during the OMV phase, and show that this conversion is sound. We may also need to separately prove that the abstractions of data structures are sound. In Chapter 4, we have described that TCService uses a linked-list data structure to implement the measurement table. Our validation aim will be to prove that in the context of TCService operations on this linked-list, abstracting it as an unbounded array in UCLID is sound. One possible way is to use a program verifier such as Boogie [18]. To perform validation on cryptographic primitives, we may again need to build a separate model, using specialized tools for cryptographic reasoning, for validation.

Chapter 7

Conclusion

In this thesis, we have presented the first formal model of CloudProxy. We use 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. The remaining properties are formalized and proved in our model. A valuable contribution of this effort is a formal API-level specification for future implementations of CloudProxy. We have also uncovered a few unintended assumptions made by the developers of CloudProxy.

7.1 Ongoing and Future Work

Model validation is a crucial component in our assurance cases, as the proven properties are only as meaningful as how sound our model abstraction is. We are exploring a model validation technique that can prove that our model encodes all the behaviours allowed by the implementation. Currently, we are working on validating our UCLID model using the techniques proposed in [26].

During modeling, tracing the counterexamples and deriving new (and non-contradicting) assumptions can be tedious. Therefore, one possible future work is to automatically generate the weakest assumption given some counterexample traces.

Axioms, which in general are implications, may have universal quantifiers in the antecedents. Proving properties which have such axioms in them may cause UCLID's quantifier instantiation heuristics to run out of memory. This is particularly evident in cases which there are more than one variable within the scope of the universal quantifier in the antecedent. Hence, another possible future work is to look into this problem and perhaps come up with some new techniques or heuristics to solve this out of memory issue.

Reasoning over cryptographic primitives may require specialized tools such as ProVerif [2]. For properties that require such reasoning, we plan to model and prove them using such tools. Our evidences in our assurance case will then involve multiple models and formal

verification techniques. In this case, Evidential Tool Bus [24] will be a useful tool in managing these various formal tools.

Bibliography

[1] *Adelard: ASCAD The Adelard Safety Case Development (ASCAD) Manual*. 1998.

[2] *An automatic cryptographic protocol verifier*. `http://proverif.rocq.inria.fr/`.

[3] T.S. Ankrum and A.H. Kromholz. "Structured assurance cases: three common standards". In: *High-Assurance Systems Engineering, 2005. HASE 2005.* Oct. 2005, pp. 99–108.

[4] Clark W. Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. 2009, pp. 825–885.

[5] Bryan Brady. "Automatic Term-Level Abstraction". PhD thesis. EECS Department, University of California, Berkeley, 2011. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-51.html`.

[6] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. "Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions". English. In: *Computer Aided Verification*. Vol. 2404. Lecture Notes in Computer Science. 2002, pp. 78–92.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: OSDI'08. San Diego, California, 2008, pp. 209–224.

[8] M.R. Clarkson and F.B. Schneider. "Hyperproperties". In: *Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st*. 2008, pp. 51–65.

[9] D. Dolev and Andrew C. Yao. "On the security of public key protocols". In: *Information Theory, IEEE Transactions on* 29.2 (1983), pp. 198–208.

[10] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *IEEE Symposium on Security and Privacy*. 1982, pp. 11–20.

[11] *GSN Community Standard Version 1*. Nov. 2011. URL: `http://www.goalstructuringnotation.info/`.

[12] Liang Gu et al. "CertiKOS: A Certified Kernel for Secure Cloud Computing". In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys '11. Shanghai, China: ACM, 2011, 3:1–3:5.

[13] J. Alex Halderman et al. "Lest We Remember: Cold-boot Attacks on Encryption Keys". In: *Commun. ACM* 52.5 (2009), pp. 91–98.

[14] Jonathan Herzog. "A Computational Interpretation of Dolev-Yao Adversaries". In: *Theor. Comput. Sci.* 340.1 (June 2005), pp. 57–81.

[15] Eunkyoung Jee, Insup Lee, and Oleg Sokolsky. "Assurance Cases in Model-Driven Development of the Pacemaker Software". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Vol. 6416. Springer Berlin Heidelberg, 2010, pp. 343–356.

[16] Tim Kelly and Rob Weaver. "The Goal Structuring Notation — A Safety Argument Notation". In: *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*. 2004.

[17] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Symposium On Operating Systems Principles*. ACM, 2009, pp. 207–220.

[18] K. Rustan M. Leino. *This is Boogie 2*. 2008.

[19] K. Rustan M. Leino and Rajeev Joshi. "A semantic approach to secure information flow". In: *LECTURE NOTES IN COMPUTER SCIENCE*. Lecture, 1997.

[20] John Manferdelli, Tom Roeder, and Fred Schneider. *The CloudProxy Tao for Trusted Computing*. Tech. rep. UCB/EECS-2013-135. University of California, Berkeley, July 2013. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.html`.

[21] Bryan Parno. "Bootstrapping Trust in a "Trusted" Platform". In: *Proceedings of the 3rd Conference on Hot Topics in Security*. HOTSEC'08. San Jose, CA, 2008, 9:1–9:6.

[22] Thomas R. Rhodes et al. *Software Assurance Using Structured Assurance Case Models*. Tech. rep. National Institute of Standards and Technology (NIST), May 2009. URL: `http://www.nist.gov/customcf/get_pdf.cfm?pub_id=902688`.

[23] John Rushby. "Proof of Separability—A Verification Technique for a Class of Security Kernels". In: *Proc. 5th International Symposium on Programming*. Vol. 137. Lecture Notes in Computer Science. Turin, Italy: Springer-Verlag, Apr. 1982, pp. 352–367.

[24] Natarajan Shankar. *Building Assurance Cases with the Evidential Tool Bus*. Mar. 2014. URL: `http://chess.eecs.berkeley.edu/pubs/1061.html`.

[25] Jr. Stephen Blanchette. *Assurance Cases for Design Analysis of Complex System of Systems Software*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, Apr. 2009. URL: `http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=29062`.

[26] Cynthia Sturton et al. "Symbolic Software Model Validation". In: *Proceedings of the 10th ACM/IEEE International Conference on Formal Methods and Models for Codesign*. Oct. 2013.

[27] Charles B. Weinstock and John B. Goodenough. *Towards an Assurance Case Practice for Medical Devices*. Tech. rep. CMU/SEI-2009-TN-018. Software Engineering Institute, Carnegie Mellon University, Oct. 2009. URL: `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8999`.