A Verified Architecture for Trustworthy Remote Attestation

Grant Jurgensen

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Committee:

Dr. Perry Alexander: Chairperson

Dr. Drew Davison

Dr. Matthew Moore

Date Defended

The Thesis Committee for Grant Jurgensen certifies
that this is the approved version of the following thesis:

A Verified Architecture for Trustworthy Remote Attestation

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Acknowledgements

This research was made possible in part due to funding from the Defense Advanced Research Project Agency, as part of the Cyber Assured Systems Engineering project.

I'd like to thank my fellow students, and particularly the participants of the lambda reading group, for the invaluable discussions which greatly accelerated my understanding of type theory and logic. Most of all, I'd like to thank my advisor, Dr. Perry Alexander for his continuous guidance, spanning both my graduate and undergraduate career. This thesis would not have been possible without his support.

Abstract

Remote attestation is a process where one digital system gathers and provides evidence of its state and identity to an external system. For this process to be successful, the external system must find the evidence convincingly trustworthy within that context. Remote attestation is difficult to make trustworthy due to the external system's limited access to the attestation target. In contrast to local attestation, the appraising system is unable to directly observe and oversee the attestation target. In this work, we present a system architecture design and prototype implementation that we claim enables trustworthy remote attestation. Furthermore, we formally model the system within a temporal logic embedded in the Coq theorem prover and present key theorems that strengthen this trust argument.

Contents

Chapter 1

Introduction

1.1 Overview

When one computer system communicates with another, it often wishes to be assured of the foreign party's *trustworthiness*. We say that system A finds system B trustworthy if A possesses evidence indicating desirable properties of B's state and/or identity. This set of properties is context-specific, and is determined by A based on the nature of the planned communication. This may include establishing the authority or authenticity of system B's components, or B's autonomy with respect to malicious third parties.

When system B constructs such evidence and presents it to A, we call this process *attestation*. The dual process, *appraisal*, is accomplished by A when it inspects the evidence and makes a trust judgment. Furthermore, we call the system performing attestation the attester, and the system performing appraisal the appraiser. In practice, a single system may alternate between the two roles. Therefore, these descriptors apply only to particular contexts.

Attestation is a broadly defined process, subsuming many existing protocols

and interactions. Identity attestation in particular is ubiquitous among internet protocols. For instance, the authentication performed during HTTPS may be viewed as an attestation, where the server attests to its identity as the owner of the domain, leveraging its digital certificate from a trusted certificate authority. In this example, the end-user's browser performs appraisal by verifying the authenticity of the supplied certificate.

Unifying such a large and diverse class of protocols under the singular umbrella of attestation allows us to consider their potential compositions, as well as generalized notions of protocol negotiation, execution, and analysis. To this end, we make use of Copland, a domain-specific language (DSL) for defining and executing attestation protocols [5, 15–19].

Attestation and appraisal can be performed locally if both the attester and appraiser are hosted by the same system. In such cases, the appraiser has the advantage of pre-existing knowledge of the attesting system, and perhaps opportunities to directly oversee attestation. However, most cases of attestation involve two physically separated parties, which we call *remote* attestation.

1.2 Architectural Considerations

The ability of a system to produce trustworthy attestation evidence is highly dependent on its underlying components and architecture. A system which is unable to strongly protect the integrity of its measurement procedures and the confidentiality of its attestation-related credentials will produce weak evidence, in that an appraiser will find the evidence largely unconvincing.

1.3 Attestation and Verification

Attestation establishes trust dynamically and in real-time. Furthermore, it often does so based on inherently probabilistic and temporal methods. Alternatively, one may establish trust in software through formal verification. In contrast to attestation, such trust is attained statically at development time, and the proofs are highly trusted, depending only on the consistency of the logic and the soundness of its implementation. Formal verification is therefore often seen as producing stronger trust results. However, its full application is often impractical, and reserved only for the most critical use cases.

Despite these disparate characteristics, the two approaches are not at odds. Rather, they may be employed in tandem to maximal effect. Instead of attempting to verify an entire system, one may instead verify a select subset of system components constituting the attestation infrastructure. Generally such attestation components are smaller than the components which they measure, and therefore more amenable to formal verification. Furthermore, portable attestation components would not need to be re-verified when transported to a new system.

1.4 Contributions

We present a formally verified system architecture designed to foster trustworthy attestation. This architecture is designed to accommodate a wide array of existing systems, which may be integrated with relatively minimal effort. We then present an embedding of a temporal logic into the general-purpose logic of the Coq theorem prover. Finally, we present the formal model of the attestation architecture, concluding with a survey of our proofs of temporal propositions over

said model.

Chapter 2

Background

2.1 The Copland Language

Copland is a DSL for attestation protocols, designed by the System-Level Design Group (SDLG) at KU, along with partners at MITRE, JHUAPL, and the NSA. The goal of Copland is to provide a standard means of defining larger attestation protocols from measurement procedures and cryptographic primitives. Copland protocols are evaluated in the context of some system state, and with respect to some initial evidence. The full protocol grammar is shown in Figure 2.1, and formal semantics have been ascribed in previous works [16, 18].

$$
\begin{aligned}
t &::= a \mid \texttt{@}p\ \texttt{[}t\texttt{]} \mid t\ \texttt{->}\ t \mid t\ s\texttt{<}s\ t \mid t\ s\texttt{\~{}}s\ t \\
a &::= \texttt{\_} \mid \texttt{!} \mid \texttt{\#} \mid \{\} \mid name\ \overline{x} \\
s &::= \texttt{+} \mid \texttt{-}
\end{aligned}
$$

Figure 2.1. The Copland protocol language. The t rule defines a top-level protocol term. p represents a place, formalized as a natural number. *name* is an identifier corresponding to an external ASP. Finally, $\overline{x}$ is a list of ASP arguments.

The a rule of the grammar defines Copland's atomic terms. Proceeding left-to-right: the `_` term is read "copy", and acts as an identity over the input evidence; the `!` term performs a cryptographic signature over the evidence; the `#` term hashes the evidence; {} returns empty evidence; and finally, an identifier followed by arguments denotes an external measuring procedure, referred to as an attestation service provider (ASP) [4]. The Copland language is extensible through the addition of custom ASPs, and transforms this otherwise pure language into a stateful one.

The t rule of the grammar defines top-level protocol terms. Once again, we proceed left-to-right. The first term a describes the aforementioned atomic terms. Next are compound terms. `@`p `[`t`]` signifies remote dispatch. The term t is sent to and executed in place p. Places and remote dispatch will be explained by way of example in Section 2.2. t `->` t describes linear sequencing. The term to the left of the arrow is executed first, and operates over the initial evidence. The output evidence is then used as the input evidence for the term to the right of the arrow. It may therefore be viewed as a composition operator. Next, the term t s`<`s t describes another sequential operation which, like the arrow operator, evaluates the right term strictly after the left. However, instead of piping the input of the first term to the second, the initial evidence is optionally passed to both subterms. The s rule defines whether the initial evidence is passed to the correspond subterm. The first s corresponds to the first subterm, and the second to the second subterm. `+` indicates that evidence is passed, whereas `-` gives the respective subterm the empty evidence. Finally, the term t s`~`s t denotes parallel branching. The evidence is passed as in the previous term, but there are no constraints on the order of execution between the two subterms. The inclusion of

parallel branching does not make the language more expressive, but rather allows for optimization as well as self-documentation when the order is insignificant.

In addition to the Copland protocol language, we also specify a Copland evidence language. The details of Copland evidence have been specified in previous work [15]. For our purposes, it suffices to know that both ASPs and the primitive cryptographic operations produce byte strings, and that Copland evidence values are a structured collection of byte strings reflecting the original protocol.

2.2 Attestation Patterns

While Copland allows for the specification of arbitrary attestation protocols, in practice we see certain recurring archetypes emerge of common attestation idioms, often associated with specific shapes of Copland protocols. A key identifying feature of these idioms is the Copland notion of *place*.

A Copland *place* is an abstract location, typically associated with a Copland *Attestation Manager* (AM) [15], a multi-purpose program capable of both attestation and appraisal. An attestation protocol that is appraised in the same place that it is evaluated is said to be an instance of *local* attestation. In contrast, a *remote* attestation occurs when attestation is performed at one place, and appraised in another.

In Copland, a top-level term is executed locally in the current place. However, we may explicitly write a remote attestation protocol, using the `@` term. For instance, consider the following protocol:

```
@p1 [(hashFile "/etc/passwd") -> !]
```

The bracketed sub-protocol requests a signed hash of the `/etc/passwd` file. Rather than being executed locally, it is instead sent to and executed at the

remote place `p1`.

This protocol raises an important question: will place `p1` really be willing to give us a hash of this sensitive file? It certainly would not be surprising if such an attestation request were rejected, especially if the requester is unknown to the attester. However, it is also conceivable that the attester would agree to this attestation. The hash operation does not reveal the files contents directly; it may only confirm whether the file matches the appraiser's expectations (by comparison to the hash of an expected exact-equivalent file).

Beyond the attester's concern of confidentiality is also its concern of expense. Some attestation protocols may take considerable system resources, and an attester is unlikely to submit to such attestations frequently. To address these problems prior to attestation, the two parties first complete *negotiation*. We assume here that all attestation protocols have been pre-agreed upon during the negotiation phase.

Returning to the topic of places, consider this new protocol:

```
(hashFile "/etc/passwd") +~+ @p1 [
    (hashFile "/etc/passwd") -> !
]
```

Here, we see an intermixing of local and remote attestation. The same file is hashed at the local place as well as the remote place `p1`. Such a protocol could be used by an appraiser to ensure that the two files are the same between the local place and `p1`. Since the local and remote sub-protocols are logically independent, we join the two with the parallel operator rather than the sequential operator. Now consider an even more complex example:

```
(hashFile "/etc/passwd") +~+ @p1 [
```

```
    ((hashFile "/etc/passwd") -> !) +~+ @p2 [
        (hashFile "/etc/passwd") -> !
    ]
]
```

This extension performs the file hash yet again, at place `p2`, demonstrating the ability in Copland to arbitrarily nest these remote dispatches. Note a subtle property about the remote dispatch of the sub-protocol to place `p2`: the term is included in the sub-protocol sent to `p1`, meaning that it will be the AM at `p1` that makes the dispatch to `p2`, not the local AM. This is often an important distinction. Perhaps `p2` does not trust the local place enough to agree to an attestation, but will negotiate with `p1`. Or perhaps `p2` is entirely inaccessible to the local place, but reachable indirectly through `p1`.

It is not necessary that a place refer to an entire system. Sometimes, a system will be partitioned into multiple layers, each considered its own place, and each with its own AM. Such a system opens the door to *layered* attestation protocols [5]. Layered attestation protocols make use of the nested dispatch idiom of the previous example to chain and sequence evidence from across multiple layers.

Most often, layered systems are structured such that the lower layers are intrinsically more trustworthy than higher layers, owing to the reduced accessibility and complexity of lower layers. To conduct a layered attestation, we make a typical attestation request to the target place, composed with a request to the layer below it. The goal is for the more trustworthy layer to provide sufficient evidence of the health of the target layer such that it effectively "extends" its trustworthiness to the target place. This process can stretch across many layers, creating arbitrarily long chains of trust [18].

For a concrete example of a layered system, consider a virtual machine monitor, running a single virtual machine. We may choose to consider the virtual machine and its monitor to be two distinct places. Since the virtual machine is logically contained within the monitor, we say that the two places form two layers of the system, with the monitor being the lower of the two layers. If we so desired, we could further subdivide the system, separating the guest operating system's kernel and user space into distinct layers.

The final attestation pattern to note is not a matter of *where* an attestation occurs, as in the previous schemas, but *when*. For transient interactions, a single attestation may be sufficient. However, for prolonged interactions, we must consider a strategy of *periodic* attestation. In any adequately complex system, attestation is necessarily limited to probabilistic notions of trust. Even if we assume a given attestation protocol is perfectly descriptive of the system at the specific moment of its execution, it cannot be expected to provide sufficient information for arbitrary future states. By the time an appraiser receives attestation evidence, the moment it was collected has already expired, and the underlying state of the attestation target has possibly already diverged from the evidence which is now being inspected. An appraiser will build trust with this evidence all the same, based on the assumption that a system is unlikely to diverge significantly over short durations. However, as more time elapses since the evidence was gathered, this trust slowly wanes, as the probability of a meaningful divergence increases.

An appraiser is motivated to request as strong and as frequent attestations as it is permitted in order to maximize its trust. The attester on the other hand will prefer to minimize its work obligation. This has a specific manifestation within layered attestation. Most often, a *deep* attestation, that is, an attestation request

over multiple layers of the system, will be stronger but more resource-intensive. In contrast, a *shallow* attestation is weaker, but also cheaper.

The competing interests of the two parties are once again resolved by compromise in the pre-attestation negotiation phase. In the context of layered attestation, the systems will likely agree to perform relatively frequent shallow attestations, and less frequent deep attestations.

Chapter 3

The Architecture

We present an architecture, referred to hereafter as the "attestation architecture", that enables trustworthy remote attestation. The attestation architecture is intended to be a flexible and generic template into which an existing system may be integrated and thereby augmented with powerful attestation capabilities. In this section, we discuss the high-level design of the architecture, the prototype implementation, and finally, a specific application of the attestation architecture within the DARPA Cyber Assured Systems Engineering (CASE) project.

3.1 Design

Before an appraiser can make a trust decision based on attestation evidence, it must determine the extent that it trusts the evidence itself. The appraiser may be convinced of the integrity of the evidence based on the Copland protocol, coupled with known properties of the target system's architecture.

There are several possibilities the appraiser must consider. First is whether the provided evidence was collected from the genuine attestation target. To this end,

most Copland protocols will terminate each sub-protocol at a given place with the signature operator `!`, and use a nonce as the initial evidence in order to ensure evidence freshness. Under such protocols, evidence can be totally identified with the target place, up to confidentiality of the private key. Assuming well-written protocols with optimally chosen signature points, the problem is reduced to the confidentiality of the foreign AMs' private keys, as determined by the respective system architectures.

Beyond the confidentiality of private keys, the appraiser must consider the integrity of the measurements. A measurement might be conducted by the genuine attestation target, but the measurement procedure tampered with by a sufficiently privileged malicious actor on the system. A highly privileged actor could conceivably circumvent detection even without affecting the measurement procedures actively, but rather by obscuring its presence totally to conventional observation. For instance, if an attacker managed to install a rootkit onto the attestation target, said rootkit could certainly hide itself from any measurement conducted from within the compromised kernel.

This leads us to our second concern: separation. We would like to ensure that our measurement functionality is adequately separated and independent of any vulnerable components. A very strong architectural candidate would therefore be a *separation kernel*, a specialized operating system kernel designed to maximize our assurance of separation [20]. However, contrary to most separation kernels, we don't want every component completely isolated, since it is important that each measurer retains one-way access to the measurement target.

Certainly if we developed a system from the ground up on such a kernel, we would have a very strong foundation to perform trusted attestation. However,

such an architecture would not accommodate the vast majority of existing systems that are overwhelmingly developed on top of general-purpose operating systems, and would need to each be painstakingly ported. Instead, we propose a generic solution where many existing systems may be integrated.

To balance these concerns, we present a stratified architecture supporting layered attestation. The foundation of our architecture is the seL4 microkernel [10]. seL4 is compelling for our high-trust argument for two reasons. First, seL4 makes extensive use of formal verification, including strong separation semantics and a proof that the kernel may operate as a separation kernel under proper configuration [13]. Second, seL4 gives us the freedom to stray from strict separation when need be, in order to give measurers one-way access to other components.

We use the CAmkES framework [12] to organize our seL4 architecture into logical components. CAmkES components, as well as their communication channels, are statically defined. All components are strongly separated from one another. Likewise, communication channels are strongly separated from all components which were not granted access under the static configuration. seL4 enforces this separation by way of virtual memory management.

One CAmkES component within the seL4 layer is a virtual machine manager, hosting a Linux virtual machine. This Linux environment constitutes our second layer. The intention is that existing Linux systems may be moved to this virtual machine with relative ease, especially in comparison to the prospect of porting a full system to a different operating system.

Now, we build out the attestation capabilities of the system by augmenting each layer with a Copland AM. To differentiate the two, we refer to the Linux AM running within the VM as the *UserAM*, and the AM running at the seL4 layer

as a CAmkES component as the *PlatformAM*. This architecture is portrayed in Figure 3.1.

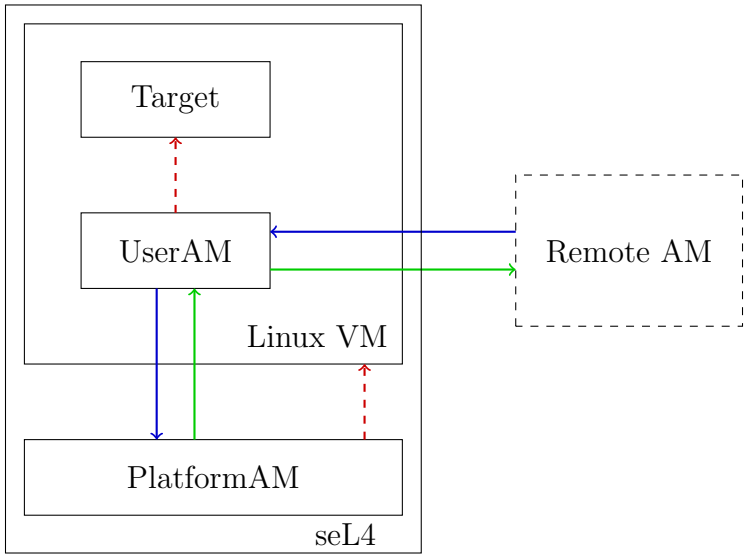

Figure 3.1. The attestation architecture. Red dashed arrows represent measurement. Blue arrows represent incoming Copland attestation requests. Green arrows represent outgoing Copland evidence.

The two AMs each have a distinct role in the system owing to their respective positions. The UserAM can perform a much wider array of measurements and more easily, because it shares the Linux environment with the attestation target, and therefore has access to the typical Linux interfaces to inspect the filesystem, other processes, etc. However, it is also also vastly more prone to corruption, again owing to its position within the corruptible Linux environment, where a malicious actor may conceivably attain equal or greater privileges. The PlatformAM, on the other hand, is strongly separated from the virtualized Linux environment, and significantly less corruptible. However, it is more difficult for the PlatformAM to conduct fine-grained measurements from outside the Linux kernel. While it has a complete view of the guest kernel space, it would require considerable effort and

re-engineering to replicate much of what is accomplished trivially by the UserAM.

As discussed in Section 2.2, a common idiom emerges from such an architecture, whereby the attester will agree with the appraiser to perform frequent shallow attestations and less frequent deep attestations. The precise periods are determined in the negotiation process according to the resource cost and the strength of the attestation protocols.

3.2 Implementation

We have developed a prototype implementation for much of the attestation architecture presented here. This includes both the the CAmkES configuration which defines the logical components and their communication channels, as well as the development of the Copland AM for both CAmkES and Linux. We have developed a collection of basic measurement procedures for the AM, and we have also defined a start-up procedure to safely release private attestation keys.

3.2.1 The Copland AM

A number of Copland AMs have been developed. We refer here to one particular implementation [9] written in CakeML and C, based on the reference implementation specified in Coq [15].

CakeML is a near-subset of Standard ML with a formally verified compiler [11]. Therefore, it brings us closer to our goal of a formally verified attestation solution. The CakeML standard library is currently immature compared to mainstream languages. However, the language includes a foreign function interface (FFI) to C that we use to add arbitrary interfaces to any necessary system calls. We also use the FFI to re-use C libraries that would be prohibitive to reproduce within

CakeML directly. For instance, we include the formally verified cryptographic library HACL* [2], written in C, through the CakeML FFI.

Beyond the verification benefits of CakeML, the language is also impressively portable. Its only dependency is the standard C library. There exists a companion to the standard library with OS-specific FFI definitions, but these definitions may easily be ported for new environments. Despite being a garbage-collected language, CakeML does not require a separate runtime environment. Instead, the garbage-collection procedure is compiled into each binary. CakeML programs may be compiled into a static binary, and due to its high interoperability with C, is often usable in any context C is used, although it may require some custom shim code.

Due to this portability, the same core AM is used for both the UserAM and the PlatformAM, the former living in a Linux environment, and the latter at the seL4 level as a CAmkES component. The core Copland interpreter, due to its purity, is identical between the two. We need only switch out the OS-specific wrappers to standard system calls, or disable some features entirely if not supported by the target environment.

3.2.2 Communication

The UserAM and PlatformAM must communicate in order to support layered attestation. The UserAM is capable of typical network communication, listening for Copland attestation requests on a predetermined port. The PlatformAM, however, has no network capabilities. The seL4 environment in which it operates is feature-sparse, and does not offer networking functionality out-of-the-box. In fact, this sparsity may be considered a feature in and of itself. The intention is for

the PlatformAM to be as strongly separated as possible from the external world to avoid malicious interference. The PlatformAM is therefore only accessible via the UserAM, through a well-defined, static communication channel spanning the VM boundary.

The limited accessibility of the PlatformAM raises the question of whether we are concerned with denial-of-service attacks against the attestation infrastructure. For instance, a compromised UserAM may choose not to forward attestation protocols to the PlatformAM. Ultimately, we do not to prioritize resilience to this class of attack. The chief goal of attestation is to provide trustworthy evidence. Denial-of-service attacks do not threaten the integrity of evidence, only its availability. Therefore, it does not threaten to provoke a false positive in appraisal. If the attestation process is interrupted by some denial-of-service attack and unable to complete, than the appraiser may safely consider the target untrustworthy until service is restored.

Communication between the PlatformAM and the UserAM occurs over a CAmkES "dataport", a static communication channel consisting of shared memory. Access to this shared memory is configured at compile-time, and given only to the two components. Memory access is enforced using seL4's usual separation methods, based on virtual memory management. The communication channel is further augmented with "events", passive general-purpose signals, used to indicate when one party has finished writing or consuming data. The dataport is shared between the PlatformAM and the seL4 virtual machine manager (VMM). The VMM, in turn, provides an interface for the guest machine to interact with this dataport. The guest Linux image is extended at compile-time with a custom kernel module that will expose a file-like object to the system, representing the

dataport. Internally, it handles all read/write calls to the dataport by making the corresponding hypercalls to coordinate with the VMM.

At the seL4 level, dataports are statically defined entities, where the various components have statically defined privileges. Thus, we know that only the PlatformAM and the VMM have read and write access to their shared dataport. However, once it is mapped into the Linux environment, we are reliant on Linux's dynamic privilege control mechanisms over files to retain the UserAM's exclusive access to the communication channel. To this end, the dataport file is assigned to a user and group by the name of "useram". We modify access controls over the file to allow just read and write access from the owner, and we assign no access to non-owners. Finally, the useram user is not a regular user; one cannot log in to the system as "useram". Instead, the UserAM application is owned by useram, and using the set-user-id functionality, is always run with the privileges of useram.

This is a common access control idiom used by Linux daemons. This method endows the executable with certain privileges (in this case, exclusive access to the dataport file), but no others[1]. This satisfies the "principle of least privilege".

Of course, this strategy is not full-proof. Any adversary which achieves root privileges may freely modify the access controls. However, this would mean the Linux VM was deeply compromised. Therefore, a deep attestation performing a sufficiently strong measurement of the Linux guest would observe this corruption, and appraisal of the attestation evidence would ultimately fail.

[1]The exact privileges of the UserAM will vary depending on the attestation requirements of the system. For instance, some measurements may require considerable privileges, such as the ability to view another process's memory space.

3.2.3 Measurement Procedures

The primary contribution of this work is in the design of the broader attestation architecture. However, some concrete measurement procedures have been written for the Copland AM. In particular, the UserAM has several measurers at its disposal. The most basic is the hashing of files and directories. Such a measurement might be useful to an appraiser who wishes to ensure that the attestation target has trusted binaries, libraries, or configuration files. In the case that we wish to dynamically measure a running process, we include a measurement procedure to read the process's address space, and hash the section we expect to be static. The measurement is comparatively brittle, and considerably more complicated than the static measurer. It would require further testing before real-world use.

At the moment, no significant measurers have been developed for the PlatformAM. However, there are several promising directions for future measurement. One particularly ambitious route would be the development of a full kernel integrity measurer, intending to detect modifications to the linux kernel obscured to the UserAM. More practical would be the inclusion of an existing measurer. While not publicly available, the Linux Kernel Integrity Measurer (LKIM) [14], developed in joint by APL and the NSA, would be one such candidate.

3.2.4 Key Release

Each AM possesses a private key it uses to endorse evidence collected during Copland protocol execution. The semantics of Copland makes the simplifying assumption that each AM has exclusive access to its private key to strongly identify evidence bundles it generates. However, in our concrete design the AMs do

not begin execution with possession of said keys. Instead, keys are strategically released during the start-up process to avoid starting a compromised AM and thereby leaking its key.

We assume that the system hardware supports some notion of a hardware-derived root of trust. For the purpose of the prototype implementation, this feature is stubbed-out. In real systems, there exist many different approaches to establishing a root of trust from the hardware, for example using a Trusted Platform Module (TPM), protected SRAM, physical unclonable functions (PUFs), or ARM TrustZone.

In order to abstract this notion of a device-specific anchor of trust, we assume that the boot process leaves some token in memory, visible to the PlatformAM, that represents an endorsement from the root of trust of the boot image. That is, the token reflects the image that was booted, and is verifiably derived from the root of trust.

Key release is performed in a "bottom-up" manner. For this system, the PlatformAM is the lowest-layered AM, and so its key will be released first. It begins execution with an encrypted key that it decrypts with the root of trust token. As a result, if an improper seL4 image was booted, potentially containing a compromised PlatformAM, then the root of trust token would be "incorrect". If the potentially compromised PlatformAM then attempts to release its key, it would fail because it decrypts its encrypted key with the wrong root of trust token.

Next, the PlatformAM prepares to release the UserAM's key. Before it does so, it must first measure the UserAM and the encompassing Linux environment to be satisfied of their authenticity and integrity. These measurements may be the same Copland protocols an appraiser might request from the attestation manager

at runtime. Once the PlatformAM is confident that the UserAM is trustworthy, it releases its key by providing it with a decryption key. This decryption key is itself derived from the PlatformAM's private key, in order to ensure that a compromised PlatformAM cannot release the UserAM's key.

Once the attestation manager's keys have been released they may start their normal procedures, waiting and listening for an appraiser's request. This key-release strategy is not expected to protect fully from leaks. Rather, it is intended to avoid leaks specifically at start-up. Over the course of the system's runtime, the system may become compromised, and a key leaked. In particular, the Linux environment is considered susceptible to runtime corruption and key leak.

The PlatformAM is considered largely immune from such leaks as a consequence of its strong separation. Furthermore, in the case of runtime credential theft within the Linux environment, we expect to be able to detect that a key was leaked, provided the PlatformAM conducts sufficiently strong and frequent measurements. In contrast, if a key is leaked during start-up, it is not necessarily possible for a component to detect the leak and publish the necessary revocation. Therefore, the prevention of key leaks during start-up is a crucial act of prophylaxis.

3.3 Application

As part of the DARPA CASE project, we had the opportunity to apply our architecture to existing systems, adding both attestation and appraisal capabilities. This experiment provided valuable insight into the practical experience of integrating a system into our architecture, as well as affording the opportunity to evaluate the strength of our resulting attestation capabilities.

The CASE project explores a scenario featuring two parties: an unmanned aerial vehicle (UAV), and a ground station. The two establish a remote connection, and the UAV flies in a path established by waypoints transmitted from the ground station. In their original forms, both systems run Linux natively, and both possess a regular Linux executable implementing the relevant flight coordination behavior. This original pair of interacting systems is depicted in Figure 3.2.

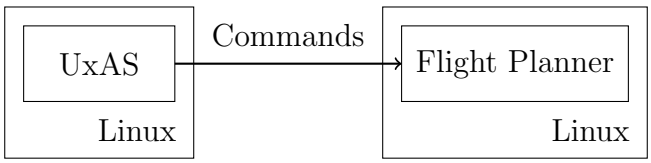

Figure 3.2. Original ground station (left) and UAV (right) architectures.

The UAV should only follow waypoints provided by a trustworthy ground station. To assure the UAV that the ground station is trustworthy, we add attestation capabilities to the ground station. In order to act on this evidence, the UAV must be modified to accommodate an appraisal process.

The ground station is modified in accordance to the previously described attestation architecture, which necessitates a move to seL4 and the inclusion of the two AMs. The UAV is modified to incorporate an AM, responsible for requesting attestation from the ground station and appraising the results. In order to avoid having to modify the flight planner software, we introduce a filter which intercepts flight-related communication. It then coordinates with the AM, and only forwards messages to the flight planner which are associated with ground stations which have recently passed appraisal. This filter provides a general solution to attestation scenarios which aim to restrict communication to legacy components to successfully appraised targets. Finally, the system is moved to seL4 to guaran-

tee separation between the filter/AM and the legacy flight planner software. Both the attestation-hardened ground station and UAV are depicted in Figure 3.3.

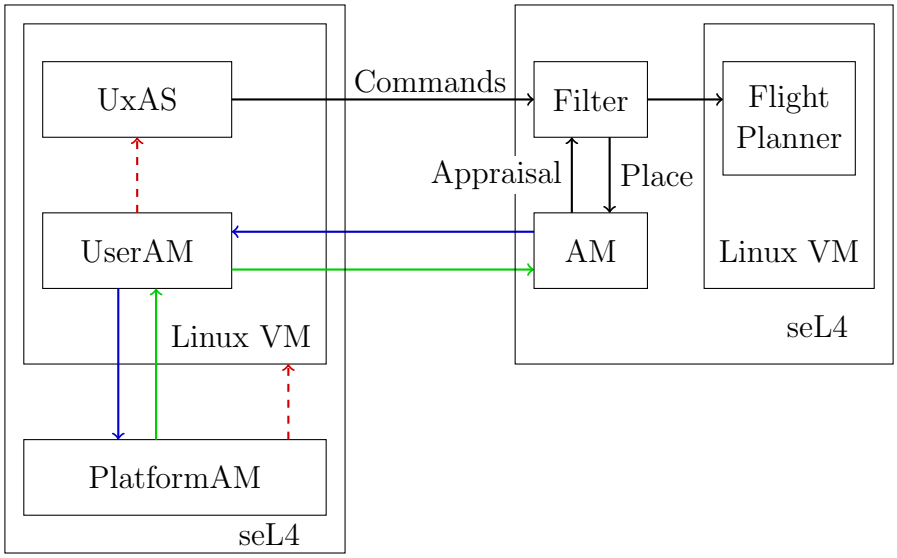

Figure 3.3. Hardened ground station (left) and hardened UAV (right). As in Figure 3.1, red dashed arrows represent measurement, blue arrows represent incoming Copland attestation requests, and green arrows represent outgoing Copland evidence.

The hardened UAV architecture has been fully implemented, and the hardened ground station has been partially implemented. There exists a prototype for the general attestation architecture [6], however the groundstation software has not been incorporated. The ground station was instead augmented with just the UserAM. Since the PlatformAM does not currently possess meaningful measurement capabilities, there were no substantial attestation benefits of applying the full attestation architecture to the ground station.

Chapter 4

Verification

4.1 Framework

The structure of the attestation architecture and the interactions of its components are carefully designed to ensure attestation is reliable and resilient to malicious interference. The goal of our verification is to formalize some subset of these properties that together establishes the trustworthiness of attestation. These properties are proven with respect to a high-level formal model of the attestation architecture.

In particular, we are concerned about the effects of malicious interference which can occur at arbitrary points during execution. To represent such possibilities, we use a branching-time logic to capture all such event traces and consider the effects on attestation.

Computation tree logic (CTL) is a temporal logic first proposed by Clarke and Emerson [3], commonly used in model checkers. CTL defines a class of formulas whose validity are considered with respect to a particular state and state transition relation. We therefore model the attestation architecture as a state transition

system, and use CTL to make assertions over the model.

Using CTL, we prove judgments of the form $R\ @s \vDash \phi$[1], called an *entailment*. Here, R is the state transition system, represented concretely as a left-total binary relation over the state type. Next, s is the current state. Finally, ϕ is a CTL proposition, also called a CTL formula. We may read this statement as "temporal proposition ϕ holds at state s, under the transition relation R".

CTL propositions may make assertions over the current state s, or they may make statements over entire *paths* or *path segments* originating from s. A path is an infinite transition sequence obeying the relation R. It has a starting state, but no final state. A path segment is also a transition sequence, but finite. That is, it has both an initial and final state. Note that, since R is left-total, every path segment is necessarily a prefix of some path.

A CTL proposition ϕ is constructed from the grammar in figure 4.1.

$$\begin{aligned}\phi ::= &\bot \mid \top \mid\ !\phi \mid \phi\ \&\&\ \phi \mid \phi\ ||\ \phi \mid \phi\ \multimap\ \phi \mid \phi\ \leftrightarrow\ \phi \mid [\![p]\!] \\ &\mid \mathrm{AX}\ \phi \mid \mathrm{EX}\ \phi \mid \mathrm{AF}\ \phi \mid \mathrm{EF}\ \phi \mid \mathrm{AG}\ \phi \mid \mathrm{EG}\ \phi \\ &\mid \mathrm{A}[\phi\ \mathrm{U}\ \phi] \mid \mathrm{E}[\phi\ \mathrm{U}\ \phi] \mid \mathrm{A}[\phi\ \mathrm{W}\ \phi] \mid \mathrm{E}[\phi\ \mathrm{W}\ \phi]\end{aligned}$$

Figure 4.1. The CTL proposition grammar. Here, p represents a predicate over states.

Our formal CTL semantics are specified by our Coq embedding [8]. Informally, we may understand CTL formulas as follows.

The first row of the CTL grammar is largely made up of familiar propositional logic constants and connectives and are assigned their typical meanings. Reading

[1]We adopt a slightly altered set of CTL notations as compared to traditional presentations. This alteration is done to allow embedding into the Coq notation system and to avoid conflicts with its existing syntax. Both the notation for entailment as well as many of the symbols in Figure 4.1 have been modified.

left-to-right, we define the false constant, the true constant, negation, conjunction, disjunction, implication, the biconditional, and finally a lifting operator which promotes arbitrary state predicates into CTL propositions. This state predicate is interpreted as applying to the current state.

The next two rows define temporal quantifiers that specify particular states and paths over which their subformulas apply. Each features two letters, and can be understood by combining the informal meaning of the first letter with the second. The first letter describes the paths under consideration, always originating from the current state. "A" may be read "for all paths", and "E" as "there exists some path". The second letter specifies the state or states in said path on which the proposition holds. "X" is short for "next", and refers to the state immediately following the current. "G" stands for "global", and refers to all states in the path. "F" stands for "finally", and asserts that the proposition holds for some state on the path. "U", short for "until", asserts that the first proposition holds until the second holds, and that the second will eventually hold. Finally, "W", sometimes called "unless" or "weak until", states that the first proposition holds until the second holds, but the second need not ever hold.

For instance, we could rigidly interpret the entailment $R\ @s \vDash \mathrm{EG}\ \phi$ according to the above rules as "there exists some path from s under transition relation R such that ϕ always holds". More simply, we may interpret the entailment as "it is possible ϕ holds forever", with an implicit understanding of the transition relation and current state which form the context of our statement. Similarly, $R\ @s \vDash \mathrm{AF}\ \phi$ is interpreted rather rigidly as "for any path from s under R, there exists some state such that ϕ holds", or more simply as "ϕ will hold at some point".

4.2 Implementation

CTL is most commonly used in model checkers, that attempt to verify properties automatically and without user input, based on internal decision procedures and heuristics. In contrast, interactive theorem provers are user-guided. Formal proof objects are generally built incrementally using *tactics*, composable procedures which solve the current goal, or reduce it to further subgoals. Theorem provers therefore put more burden on the end user to guide a proof, however they also enable the user to prove arbitrarily complex propositions, without regard for any internal proof-search limitations.

In this work, we elect to use the Coq theorem prover. Coq presents a general-purpose intuitionistic logic, which we extend axiomatically into a classical logic by adding the law of the excluded middle, as well as other commonly accepted axioms such as functional and propositional extensionality. This acts as the host logic where we embed CTL.

4.2.1 Logical Embedding

When embedding a particular logic into a theorem prover, one has the choice of developing either a shallow or a deep embedding. In a deep embedding, one would define the embedded logic constructs as a set of uninterpreted symbols. Then, one defines the entailment operator which determines which formulas are derivable.

Deep embeddings are appealing in that embedded constructs are only meaningful in the context of their entailments, and therefore there is no confusion between propositions of the embedded logic and those of the host logic. However, they may be prohibitively complicated in their entailment definitions.

In a shallow embedding, the distinction between the embedded and host logic is greatly reduced. Propositions in the embedded language are defined by direct mappings to propositions in the host language. Since embedded propositions then carry their meaning in their definitions, the definition of entailment is largely trivialized.

In this case, we elect to shallow embed CTL into Coq. Initial efforts focused on a deep embedded implementation. However, this approach proved impractical. The obvious relational definition of entailment was unsound and rejected by Coq due to technical issues involving unavoidable positive occurrences in the inductive definition of negation and implication entailments [1]. A functional definition of entailment was possible, but struggled against Coq's requirement that recursion be obvious terminating - that is, that all recursive applications occur over structurally smaller values. This led to complicated and unintuitive definitions of the CTL constructs.

Not only is the shallow embedding easier to work with, we also believe that the disadvantages of a shallow embedding are largely mitigated in practice by obscuring the underlying representation, and presenting a separate set of theorems and proof tactics through which to reason about CTL constructs.

We define the type of CTL propositions, which we call tprop, short for "temporal proposition", parameterized by the state type.

```
Definition tprop state :=
  ∀ R: relation state,
    transition R →
    state →
    Prop.
```

In the definition above, transition is a predicate which asserts that the binary

relation R acts as a transition relation, in the sense that it is left-total. With this representation of CTL propositions, the definition of entailment becomes trivial:

```
Definition tentails {state} (R: relation state) {t: transition R}
  (s: state) (P: tprop state) :=
  P R t s.
```

Our entailment definition just applies the parameters to our CTL proposition, exploiting its shallow representation. Notice the two arguments surrounded by curly braces in the definition of tentails. These indicate that the arguments are implicit. We will not explicitly instantiate tentails with these values. Rather, Coq will automatically infer their values. The first implicit argument, *state*: Type, is inferred from the other arguments. The second implicit argument, t: transition R, is inferred using Coq's "typeclass" functionality, inspired by the Haskell feature of the same name. Coq allows us to define custom notations, which we use to declare a notation for entailment in alignment with our presentation thus far[2]:

```
Notation "R @ s ⊨ P" := (tentails R s P).
```

The definitions of simple CTL propositions (i.e., those which do not quantify over paths) are largely straightforward. Consider the following definitions of the constant truth constant and of conjunction:

```
Context (state : Type).

Definition ttop : tprop state :=
  fun _ _ _ ⇒ True.

Notation "⊤" := ttop.

Definition tconj (P Q: tprop state): tprop state :=
  fun R _ s ⇒ R @s ⊨ P ∧ R @s ⊨ Q.
Notation "P && Q" := (tconj P Q).
```

[2]This notation declaration, as well as future notation definitions, are simplified slightly as compared to the real implementation to avoid extraneous detail regarding parsing precedence, print-formatting, and notation scopes.

Note the distinction between CTL and Coq formulas. $\top$ and && are CTL propositions, while True and $\wedge$ are Coq propositions.

Although our type family tprop admits any Coq relation between a transition relation and state, we choose to restrict ourselves to the pre-defined CTL formulas, in order to separate our understanding of the CTL semantics from its underlying representation. To further obfuscate the representation of these CTL formulas, we make their definitions opaque, *i.e.* we forbid the terms from being unfolded, and restrict ourselves to a set of basic theorems which capture the meaning of the CTL formulas without exposing their underlying representation. For instance, instead of directly appealing to the definitions of the top value and conjunction shown above, we use the following fundamental theorems:

```
Context (state : Type) (R: relation state) {T: transition R} (s: state).

Theorem tentails_ttop :
  R @s ⊨ ⊤.

Theorem tentails_tconj : ∀ P Q,
  R @s ⊨ P →
  R @s ⊨ Q →
  R @s ⊨ P && Q.
```

Not only are these theorems more readable than the definitions, they also force us to only consider CTL propositions in the context of an entailment, as we would in a deep embedding.

4.2.2 Coinductive Paths

Paths in CTL are infinite, as they are intended to capture an entire possible future of the state transition system. This poses a challenge to formalization, since most type definitions in Coq are *inductive*. A term of an inductive type is built from finitely many constructors, and are therefore themselves finite. From

an inductive definition arises an *induction principle*, automatically generated by Coq, which allows us to prove a proposition by structural induction (or in its more generalize form, to build a term by well-founded recursion).

To define types with potentially infinite terms, Coq provides the dual notion of *coinductive* types. Terms of coinductive types are called *codata*. While terms of inductive types are conceptually built from a "bottom-up" perspective, codata are represented by "top-down" descriptions. Informally speaking, the only constraint on coinductive terms is that they are well-defined when we destruct them.

For instance, we may define paths in the following coinductive fashion:

```
Context {state : Type} (R : relation state).
CoInductive path (s: state) : Type :=
  | step : ∀ s', R s s' → path s' → path s.
```

The step constructor defines a path originating from s by connecting a relation step $R\ s\ s'$, and a path originating from s'. Since step is the only path constructor, all paths are infinite under this representation, as intended.

This definition is appealing in that it captures our intuitive notion of a path. Unfortunately, coinductive types are difficult to reason about in Coq. While inductive types produce induction principles, there is no corresponding elimination principle for coinductive types. One must instead use the primitive notion of a cofixpoint to produce all necessary codata, that are subject to unintuitive "guardedness" conditions. Finally, we face significant obstacles reflecting infinite observations into finite ones. Even something as simple as proving two paths equivalent requires a new coinductive notion of equality specialized to the infinite structure of our paths, as well as a specialized axiom to reflect this equality to the typical propositional equality. For these reasons, we avoid an explicitly coinductive formulation of paths. Fortunately, there exists a general isomorphism from

coinductive types to function types[3]. Below is the alternative functional definition of paths:

```
Context {state : Type} (R: relation state).

Definition path (s: state) : Type :=
  Σ p: nat → state,
    p 0 = s ∧
    ∀ i, R (p i) (p (S i)).
```

In this formulation, we define a path originating from s as some function from the natural numbers to the state type, where the first element is s, and subsequent elements are related by R. Concretely, it is a dependent pair[4] of this characteristic function and a proof of the path properties.

The isomorphism between this path definition and the previous is self-evident. Instead of explicitly constructing a sequence of steps in a concrete object, we instead construct a function mapping an index to the state at that position. To capture the same structure as was in the original path definition, we further constrain the function to anchor it to the correct starting point, and ensure the pairwise relatedness of consecutive states.

Since a path is primarily identified with its underlying function, we allow the implicit coercion of the former to the latter. That is, for some path p and natural number n, we interpret the expression $p\ n$ as the state in p at index n. With this representation, predicates over paths are defined quite straightforwardly.

```
Definition in_path {s} (x: state) (p: path s) : Prop :=
  ∃ i, p i = x.
```

[3]By a type isomorphism, we mean a bijection between types. Strictly speaking, this isomorphism relies on extensionality principles for both functions and individual coinductive types, which are taken for granted in ordinary mathematics, but must be added to axiomatically to Coq.

[4]We adopt the sigma binder notation for dependent pair types. This notation is standard in type theory literature, however Coq uses alternative notation by default. The dependent pair $\Sigma\ x,\ F\ x$ is instead written $\{x \mid F\ x\}$ when F is a predicate, and $\{x\ \&\ F\ x\}$ otherwise.

```
Definition in_path_before {s} x i (p: path s) : Prop :=
  ∃ j, j < i ∧ p j = x.
```

The definitions of path-quantifying CTL formulas also follow quite directly. For instance, consider the following theorems that capture our definitions of EG and AU:

```
Theorem rew_EG : ∀ P,
  R @s ⊨ EG P =
  ∃ p: path R s, ∀ s', in_path s' p → R @s' ⊨ P.
Theorem rew_AU : ∀ P Q,
  R @s ⊨ A[P U Q] =
  ∀ p: path R s, ∃ i,
    (∀ x, in_path_before x i p → R @x ⊨ P) ∧
    R @(p i) ⊨ Q.
```

The full set of fundamental theorems may be found in the CTL repository [8], in `Ctl/Basic.v`.

4.3 Model

We distinguish between two kinds of state. The first, which we call a state *label*, describes in general terms the internal state of a particular component. This corresponds to the implicit state arising from one's current position in the program.

The second is called the *environment*. This is a mutable data store shared by all components, representing the explicit storage of values. To support the notion of secret values and non-interference, we augment this environment with a description of access controls.

Formally, we define two privileges: the privilege to read some data store, and the privilege to write to some data store.

```
Inductive privilege : Set :=
  | p_read
  | p_write.
```

An access control is defined as a decidable relation between components and privileges.

```
Definition access := comp → privilege → bool.
```

The particular definition of the component type comp is not of great interest. We use the string type for the sake of extensibility.

We apply the natural inclusion ordering over relations to access controls. That is, we say $a_1 < a_2$ if and only if a_2 is strictly more permissive than a_1. This ordering gives rise to a boolean lattice and enables compositional construction of access controls using meet and join operations.

We then declare our environment type as a partial mapping of variables to access controls and heterogeneous values:

```
Definition any : Type := Σ X, X.
Definition box {X} (x: X) : any := ⟨X, x⟩.
Definition env := var → option (access * any).
```

The type any is intended to represent a value of arbitrary type. Concretely, it is a dependent pair where the left pair element is a type, and the right element is an inhabitant of that type. The pair is logically identified with its right element. We include the box definition to construct and represent elements of any while leaving the type implicit. As with comp, the definition of the variable type var is not of great significance, and we again use the string type.

Singleton environments are represented $x \mapsto y$. Access controls default to the top element of the lattice, *i.e.* the totally permissive access control. Access controls can be overwritten using the notation a ? Γ, which assigns the access

control a to all variables in the environment Γ. Finally, two environments can be joined with the notation Γ_1 ;; Γ_2. If there is any overlap in the domain of the two environments, the combined environment uses the value under Γ_1 (the elements of Γ_1 are said to "shadow" those of Γ_2).

We now introduce propositions to describe interactions with the environment that respect our access controls.

```
Definition read {V} (Γ: env) (c: comp) (name: var) (v: V) : Prop :=
  ∃ acc: access,
    Γ name = Some (acc, box v) ∧
    acc c p_read.

Definition write {V} (Γ: env) (c: comp) (name: var) (v: V) (Γ': env) : Prop
:=
  ∃ (acc: access) curr,
    Γ name = Some (acc, curr) ∧
    acc c p_write ∧
    Γ' = acc ? name ↦ v ;; Γ.

Definition changeAcc (Γ: env) (name: var) (f: access → access) (Γ': env) :
Prop :=
  ∃ (acc: access) V (v: V),
    Γ name = Some (acc, box v) ∧
    Γ' = f acc ? name ↦ v ;; Γ.
```

Now, we may finally define our transition relation modelling the attestation architecture. The entire definition is quite large, and can be found in the attestation architecture model repository [7]. Here, we will only show a portion of the state transition of the UserAM to demonstrate the style in which we define our larger relation:

```
Inductive useram_label :=
  | useram_wait_key
  | useram_listen
```

```
  | useram_shallow_attest
  | useram_deep_attest.

Definition useram_init_env : env :=
  only "useram" ? "useram_key" ↦ encr_useram_key.

Definition decrypt_useram_key key decr_key : useram_key_t :=
  match (key, decr_key) with
  | (encr_useram_key, good_decr_key) ⇒ good_useram_key
  | _ ⇒ bad_useram_key
  end.

Definition shallow_attest (good_os good_target good_meas: bool) :=
  good_os = true → good_target = true → good_meas = true.

Inductive useram_trans : relation (useram_label * env) :=
  | useram_get_key : ∀ Γ Γ' encr_key decr_key,
      read Γ "useram" "useram_key" encr_key →
      read Γ "useram" "vmm_dataport" decr_key →
      write Γ "useram" "useram_key"
        (decrypt_useram_key encr_key decr_key) Γ' →
      useram_trans
        (useram_wait_key, Γ)
        (useram_listen, Γ')
  | useram_get_shallow_req : ∀ Γ,
      useram_trans
        (useram_listen, Γ)
        (useram_shallow_attest, Γ)
  | useram_do_shallow_attest : ∀ Γ Γ' os target meas,
      read Γ "useram" "good_os" os →
      read Γ "useram" "good_target" target →
      shallow_attest os target meas →
      write Γ "useram" "shallow_attest_result" meas Γ' →
      useram_trans
        (useram_shallow_attest, Γ)
        (useram_listen, Γ)
  | useram_get_deep_req : ∀ Γ Γ',
```

```
      write Γ "useram" "vmm_dataport" attest_req Γ' →
      useram_trans
        (useram_listen, Γ)
        (useram_deep_attest, Γ')
  | useram_wait_deep_attest : ∀ Γ (meas: bool),
      read Γ "useram" "vmm_dataport" meas →
      useram_trans
        (useram_deep_attest, Γ)
        (useram_listen, Γ).
```

4.4 Automated Proof Search

Since our transition relation is quite large, each CTL proof over the relation will also be quite large. All such proofs must be carried out by cases on the potential steps taken by the transition relation, of which there are 16. Even proofs of relatively simple CTL propositions require lengthy proofs. However, most cases are quite straightforward in practice. To overcome this tedium that degrades efficient progress and to take advantage of the computerized proof environment, we write an automated proof search tactic specialized to our domain.

Our consistent formulation of transition steps in terms of well-behaved *read* and *write* statements presents a specialized domain from which we can consistently and mechanically deduce significant insights. For instance, we have the following theorems concerning *read* statements:

```
Theorem no_lookup_no_read {Γ c x V} {v: V}:
  Γ x = None → ¬ read Γ c x v.

Theorem wrong_lookup_no_read {Γ acc c x V} {w v: V}:
  Γ x = Some (acc, box w) →
  w ≠ v →
  ¬ read Γ c x v.
```

```
Theorem good_lookup_read {Γ acc c x V} {w v: V}:
  Γ x = Some (acc, box w) →
  read Γ c x v →
  w = v.
Theorem good_lookup_read_acc {Γ acc c x W V} {w: W} {v: V}:
  Γ x = Some (acc, box w) →
  read Γ c x v →
  acc c p_read.
```

Note that each theorem follows from an assumed equality between $\Gamma\ x$ and some canonical value. Given some assumption read $\Gamma\ c\ x\ v$ in our proof state, we may then attempt to compute $\Gamma\ x$, and if it reduces to a canonical term, we then deduce new facts from the relevant theorems.

Once we deduce all relevant facts from the available read, write, and changeAcc assumptions, we may then employ conventional proof search techniques to resolve the proof goal. We call this tactic `rw_solver`, that abbreviates "read-write solver". For simple CTL propositions making straightforward assertions over the environment, `rw_solver` often solves all but one or two of the 16 cases produced by case analysis over a step of the attestation architecture transition relation.

4.5 Theorems

We conclude with a survey of the most significant propositions proven over the attestation architecture model. First we prove that the PlatformAM's private key is never compromised under our model, in the sense that only the PlatformAM ever possesses read access.

```
Definition platam_key_secure : tprop attarch_state := ⟦fun '(_, Γ) ⇒
  ∀ c key_t (key: key_t),
    read Γ c "platam_key" key →
```

```
    c = "platam"
]].
Theorem platam_key_uncompromised: ∀ s0,
  attarch_trans @s0 ⊨
    is_init_state ⤳
    AG platam_key_secure.
```

Note that the state predicate is_init_state does not enforce that the state specifically reflects a good or bad system. It only asserts that the state describes a system in the boot phase with a minimal environment.

We do not present the full proofs here, but each is available in the repository [7], and we shall comment on the general structure. For this theorem, we reflect the AG construct into a statement over the reflexive transitive closure of the transition relation. The proof then proceeds by induction over the structure of the transitive closure witness. This produces 16 subcases, 14 of which are resolved in a single step by our domain-specific automation tactic, `rw_solver`. The remaining two are straightforward.

We also prove that the PlatformAM can only possess its genuine private key when the system is operating from a good initial image.

```
Definition platam_key_good : tprop attarch_state := [[fun '(_, Γ) ⇒
  ∃ acc, Γ "platam_key" = Some (acc, box good_platam_key)
]].
Theorem platam_good_key_good_image : ∀ s0,
  attarch_trans @s0 ⊨
    is_init_state ⤳
    AG (platam_key_good ⤳ image_is_good).
```

The general structure of this proof is the same as the previous. However, we do rely in one subcase on an external lemma, which states that when the boot token is good, then the boot image must be good. This lemma is proved separately in

much the same style as the theorems presented so far.

Similar to our first theorem, we prove a proposition asserting that the UserAM's private key is uncompromised. However, we only prove that this is case during the start-up process. After this point, the status of our key is much less certain.

```
Definition useram_key_secure : tprop attarch_state := ⟦fun '(_, Γ) ⇒
  ∀ c,
    read Γ c "useram_key" good_useram_key →
    c = "useram"
⟧.

Definition useram_key_released : tprop attarch_state := ⟦fun '(l, _) ⇒
  ∃ pl ul,
    l = sel4_run pl (vm_run ul) ∧
    ul ≠ useram_wait_key
⟧.

Theorem useram_key_uncompromised_setup : ∀ s0,
  attarch_trans @s0 ⊨
    is_init_state ⇾
    A[useram_key_secure W useram_key_released].
```

In contrast to the previous theorems, this proposition is stated in terms of the AW connective. In this proof, we first strengthen the left AW subformula to the statement that the UserAM's private key is not the good key, which is true until the key is released. We then proceed using a custom AW introduction rule, from which the normal definition follows. This introduction rule tasks us to prove that either subformula holds on the current state, and that either subformula holds for some state at the end of a path segment, where the left subformula holds for all earlier states in the segment, and the right subformula does not. In this latter case, our automation resolves every further subgoal.

We are also able to nest the path-quantifying CTL formulas to some degree be-

fore encountering prohibitive complexity. For instance, we prove the permanence of OS corruption, expressed by the following theorem:

```
Definition os_corrupted : tprop attarch_state := ⟦fun '(_, Γ) ⇒
  ∃ acc, Γ "good_os" = Some (acc, box false)
⟧.

Theorem os_corrupted_permanent : ∀ s0,
  attarch_trans @s0 ⊨
    is_init_state ⊸
    AG (os_corrupted ⊸ AG os_corrupted).
```

Chapter 5

Conclusion

We have presented (1) the design of a system architecture intended to augment a wide array of existing systems with trustworthy attestation capabilities, and (2) a high-level formal model of the architecture in an embedded temporal logic within the Coq theorem prover. The attestation architecture has been prototyped and applied to the DARPA CASE project, where it was evaluated by an independent red team and found to prevent certain attacks which went undetected in the original architecture.

The verification effort was split between the development of a general-purpose CTL embedding, as well as a specific model built atop the CTL library capturing the behavior of the attestation architecture. The CTL embedding contains not just basic definitions, but numerous tactics and theorems to facilitate reasoning about CTL constructs. Since it is a standalone library, independent of the architecture model, it may be re-used for future verification efforts.

Finally, we developed a notion of access-controlled heterogeneous environments. By defining the rules of our attestation architecture transition relation in terms of access-control-respecting read and write operations, we are presented

with a specialized domain of proof goals. To accelerate proof development, we developed the `rw_solver` tactic to automatically deduce common observations in this domain before employing conventional proof-automation procedures to solve straightforward proof goals. This tactic greatly alleviated the tedium of large CTL proofs.

There are numerous ways that the work presented here may be extended. With respect to the prototype architecture implementation, more work may be done to integrate powerful measurers, particularly to the PlatformAM. Furthermore, we aim to eventually apply the implementation to hardware which readily supports a hardware root of trust, and to fully integrate said root of trust to enable the full chain of trust described in Section 3.2.4.

There are considerable opportunities to improve the CTL embedding. While it is intended to serve as a standalone library, many of the properties that have been proven and the tactics developed have arisen from their necessity to the attestation architecture model proofs. It is likely that there exists a number of significant properties of CTL that would be needed for other projects that have not yet been proven in our library. Given the popularity of CTL within model checkers, it would likely be possible to introduce significantly more automation to arbitrary CTL proofs, and particularly CTL formulas which span over decidable state predicates.

The access-controlled environments used by the attestation architecture model appear to be a promising and expressive abstraction. We may consider how to generalize the environmental logic in this work into a more flexible framework. Of particular interest would be the application of access controls to existing logics for reasoning about state, such as separation logic.

Finally, we may focus on optimizing our existing proof automation. The `rw_solver` tactic solves many of our proof goals. However, it may execute for several tenths of a second before either succeeding or failing. This is not an issue for a single use, but execution is further slowed when the tactic is applied to a dozen subgoals simultaneously, as we often do in our proofs. The tactic's execution time is also prohibitive to its invocation in higher-order automation contexts involving frequent backtracking.

References

[1] Coq language documentation: Positivity condition. https://coq.inria.fr/refman/language/core/inductive.html#positivity, 2021.

[2] K. Bhargavan, B. Beurdouche, J.-K. Zinzindohoué, and J. Protzenko. Hacl*: A verified modern cryptographic library. *ACM CCS*, September 2017.

[3] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In D. Kozen, editor, *Logic of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberk.

[4] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2):63–81, June 2011.

[5] S. C. Helble, I. D. Kretz, P. A. Loscocco, J. D. Ramsdell, P. D. Rowe, and P. Alexander. Flexible mechanisms for remote attestation. *ACM Transactions on Privacy and Security*, 24(4), September 2021.

[6] G. Jurgensen. An attestation architecture prototype. https://github.com/ku-sldg/attarch/tree/thesis, 2022.

[7] G. Jurgensen. A formal model of an attestation architecture. https://github.com/ku-sldg/attarch-model/tree/thesis, 2022.

[8] G. Jurgensen and A. Cousino. A shallow embedding of computation tree logic (ctl) in coq. https://github.com/ku-sldg/ctl/tree/thesis, 2022.

[9] G. Jurgensen, A. Petz, P. Alexander, T. Barclay, E. Komp, M. Neises, and A. Cousino. A copland attestation manager (am) in cakeml. https://github.com/ku-sldg/am-cakeml, 2021.

[10] 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 *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, New York, NY, USA, 2009. ACM.

[11] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM.

[12] I. Kuz, Y. Liu, I. Gorton, and G. Heiser. Camkes: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5):687–699, 2007. Component-Based Software Engineering of Trustworthy Embedded Systems.

[13] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429, 2013.

[14] J. A. Pendergrass and K. N. McGill. Lkim: The linux kernel integrity measurer. volume 32, pages 509–516, 2013.

[15] A. Petz and P. Alexander. A copland attestation manager. In *Hot Topics in Science of Security (HoTSoS'19)*, Nashville, TN, April 8-11 2019.

[16] A. Petz and P. Alexander. An infrastructure for faithful execution of remote attestation protocols. In A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez,

editors, *NASA Formal Methods*, volume 12673 of *Lecture Notes in Computer Science*, pages 268–286, Berlin, Heidelberg, 2021. Springer International Publishing.

[17] A. Petz, G. Jurgensen, and P. Alexander. Design and formal verification of a copland-based attestaiton protocol. In *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'21)*, virtual, November 2021.

[18] J. Ramsdell, P. D. Rowe, P. Alexander, S. Helble, P. Loscocco, J. A. Pendergrass, and A. Petz. Orchestrating layered attestations. In *Principles of Security and Trust (POST'19)*, Prague, Czech Republic, April 8-11 2019.

[19] P. Rowe, J. Ramsdell, and I. Kretz. Automated trust analysis of copland specifications for layered attestations. In *Principles and Practice of Declarative Programming (PPDP 21)*, Sept. 2021.

[20] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. Association for Computing Machinery.