Formally Verified Bundling and Appraisal of Evidence for Layered Attestations

Adam Petz

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 Doctor of Philosophy in Computer Science.

PhD Committee:

Dr. Perry Alexander: Chairperson

Dr. Alex Bardas

Dr. Drew Davidson

Dr. Andy Gill

Dr. Prasad Kulkarni

Dr. Emily Witt

06-08-2022

Date Defended

The PhD committee for Adam Petz certifies
that this is the approved version of the following dissertation:

Formally Verified Bundling and Appraisal of Evidence for Layered Attestations

Committee:

_____

Chairperson

_____

_____

_____

_____

_____

06-08-2022

Date Approved

Abstract

Remote attestation is a technology for establishing trust in a remote computing system. Core to the integrity of the attestation mechanisms themselves are components that orchestrate, cryptographically bundle, and appraise measurements of the target system. Copland is a domain-specific language for specifying attestation protocols that operate in diverse, layered measurement topologies. In this work we formally define and verify the Copland Virtual Machine alongside a dual generalized appraisal procedure. Together these components provide a principled pipeline to execute and bundle arbitrary Copland-based attestations, then unbundle and evaluate the resulting evidence for measurement content and cryptographic integrity. All artifacts are implemented as monadic, functional programs in the Coq proof assistant and verified with respect to a Copland reference semantics that characterizes attestation-relevant event traces and cryptographic evidence structure. Appraisal soundness is positioned within a novel end-to-end workflow that leverages formal properties of the attestation components to discharge assumptions about honest Copland participants. These assumptions inform an existing model-finder tool that analyzes a Copland scenario in the context of an active adversary attempting to subvert attestation. An initial case study exercises this workflow through the iterative design and analysis of a Copland protocol and accompanying security architecture for an Unpiloted Air Vehicle demonstration platform. We conclude by instantiating a more diverse benchmark of attestation patterns called the "Flexible Mechanisms for Remote Attestation", leveraging Coq's built-in code synthesis to integrate the formal artifacts within an executable attestation environment.

Contents

Chapter 1

Introduction

1.1 From Trust to Remote Attestation

Security decisions often depend on trust in the components or participants that make up the context of those decisions. However, the term "trust" is heavily overloaded throughout the fields of computing [23], psychology, sociology, and beyond. A useful baseline definition in computing is the one provided by the Trusted Computing Group: "An entity can be trusted if it always behaves in the expected manner for the intended purpose" [67]. A more practical refinement of the notion of trust for our purposes is outlined in Andrew Martin's *The ten-page introduction to Trusted Computing* [43], where he asserts the following requirements for a computing platform to be trusted: 1) strong identification of itself, and 2) strong identification of its current configuration and running software. This notion of trust goes beyond more shallow forms of authentication, since experience shows that a motivated attacker will compromise layered systems despite a well-intentioned user at the surface.

One technology for establishing trust in a remote computing system is *remote attestation*. Remote attestation is the activity of making a claim about properties of a target by supplying evidence to an appraiser over a network [12]. The *target* is a platform providing a useful service or protecting a sensitive resource, while an *appraiser* is a remote entity that seeks evidence of the target's trustworthiness. A simple example of such a claim would be: Target platform T is running an authentic instance of application E, where E handles sensitive data in a manner acceptable to the appraiser. Figure 1.1 diagrams a simple attestation architecture for this scenario. The motivation of the appraiser is to increase its trust in further interactions with application E running on T with respect to the sensitive data.

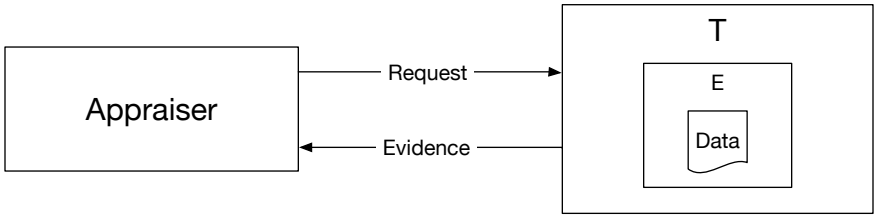

Figure 1.1. A simple remote attestation architecture: An appraiser makes a request over a network to target(T). T responds with evidence as to the trustworthiness of executable(E).

In an ideal scenario, assume E has been formally verified for functional correctness: the complete behavior of E is captured in a specification and proven correct with respect to the appraiser's expectations. Is it then sufficient to hash E's code at load-time, start its execution, and send the hash back to the appraiser? There are two immediate shortcomings of this attestation strategy. First, it says nothing about the identity of the system making the claim about E. A hash of the code proves only that *someone* knows what constitutes a valid instance of E. We can remedy this by equipping the target with a digital signature capability. Figure 1.2

shows an updated architecture for T with an authentication component AUTH. While AUTH remains abstract here, its instantiation on a real system could be a cryptographic library, stand-alone oracle, or even a hardware co-processor. It's role is to uniquely identify the platform and bind that identity to evidence. With AUTH we can authenticate a genuine hash of E, but it still remains to prove that E is in fact running on T. Still absent from this architecture is the component performing the hash of E. In the attestation literature these components are known as *measurers* [12], and are responsible for examining and reporting the configuration and operation of other components. In our simple scenario, the measurement in question is a load-time hash of E's binary.

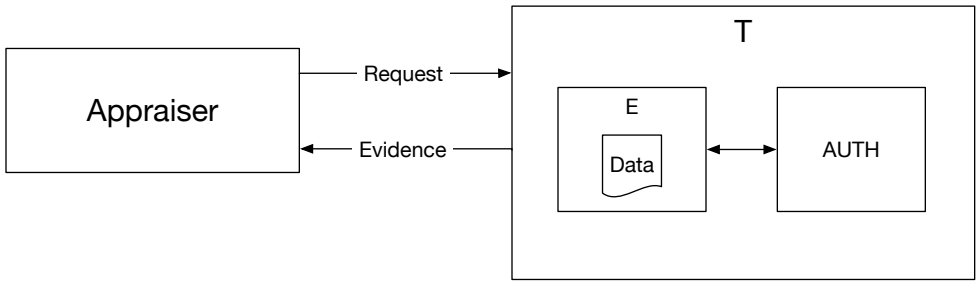

Figure 1.2. Adding AUTH signature server to target platform T.

Figure 1.3 shows an updated architecture with a measurement component MEAS. The dotted line from MEAS to E denotes a "measures" relationship. This architecture allows MEAS to hash the binary of E and authenticate it with AUTH. Until now we have ignored the other components that MEAS, AUTH, and E rely on for successful execution. Often implicit in the functional correctness argument of E is that it has a safe and sufficiently isolated context in which to run. However, experience shows that adversaries attack layers below or adjacent to the component of interest [43, 60]. Figure 1.4 shows a more realistic, layered architecture for the target T where an arrow pointing from one component to another signi-

fies a "contextual dependency" [60]. A rogue operating system or network driver could undermine the security benefits of E. It becomes apparent here that trust in MEAS and AUTH alone are insufficient to achieve trust in E.

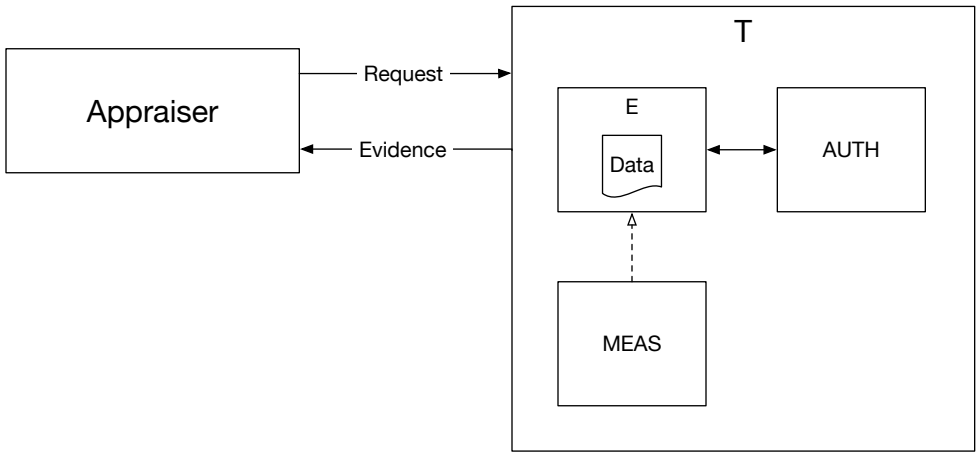

Figure 1.3. Adding Measurer MEAS to target platform T.

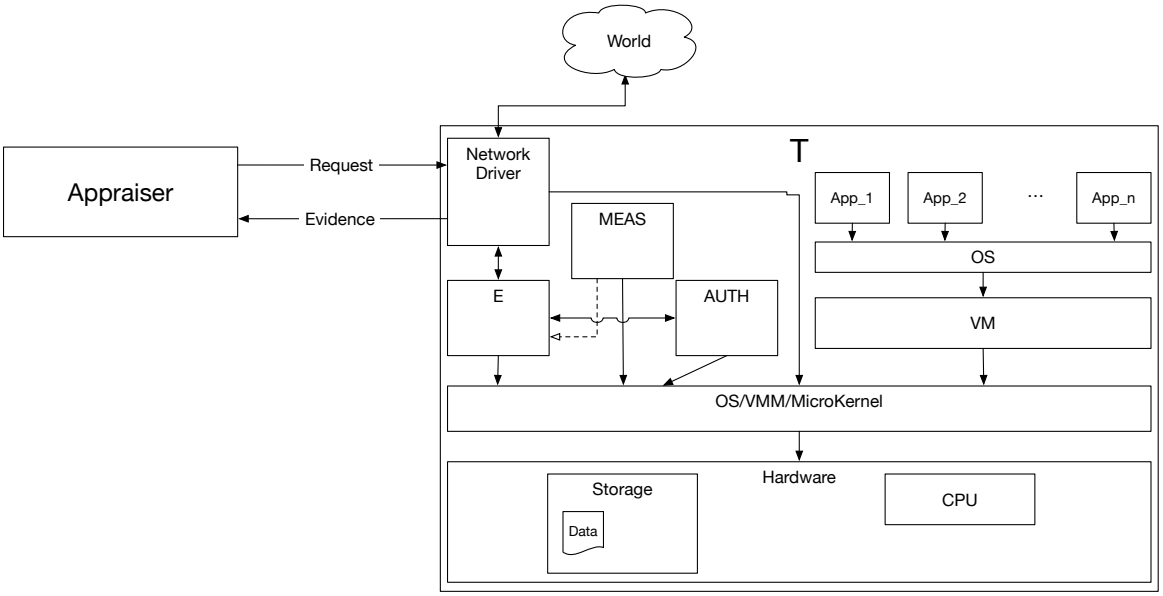

Figure 1.4. A more realistic target platform. Arrows indicate inter-component dependencies.

Another dimension of measurement in an attestation architecture is when multiple participants perform attestations on behalf of one another. Figure 1.5 shows an example architecture with multiple protocol participants. In this scenario the Relying Party engages in a delegated appraisal approach, where it specifies that the Attester should send evidence to a separate Appraiser to acquire a certificate that vouches for its trustworthiness. This topology is desirable when the attestation target does not trust the Relying Party with direct measurements that reveal its configuration, but instead trusts a third-party appraiser. In Chapter 3 we introduce a formal language that supports specifying multi-participant scenarios such as this. Further, this Certificate Style attestation pattern is merely one of a collection of such *Flexible Mechanisms* proposed by Helble et. al. [27], and in Section 7.3 we instantiate each pattern in this benchmark by leveraging the formal attestation components outlined later in this work.

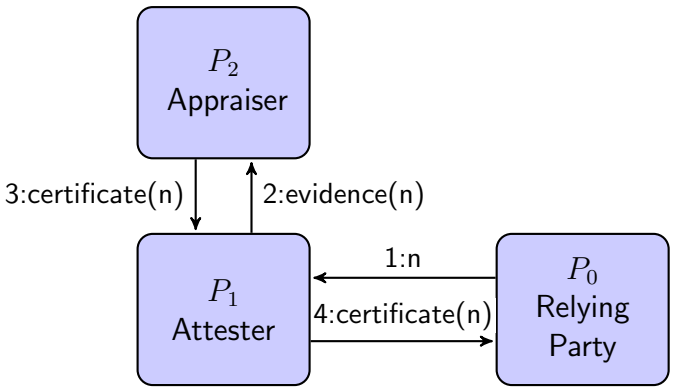

Figure 1.5. Certificate-Style. Fig. 5 on pg. 29:15 of [27].

1.2 System-Level Security

Computing systems are built from a diverse collection of hardware and software resources that interact in complex, layered ways. The security of one layer depends on the security of layers below and adjacent to it. For example, a virus checker's recommendations are useless if it depends on a malicious OS kernel or an adjacent component that can clean up after malicious actions to fool the virus scan. Even when a system starts in a good state, some applications have a sliding window of "secure dynamic states" and misbehave due to malicious inputs from a remote actor. Layers below the target application, such as OS kernels and other core system-level software, also remain vulnerable to static and dynamic corruption.

For all of these reasons, absolute system-level security–where every component is impervious to attack–is an impractical goal on any useful system. Attackers will attempt to exploit systems, and many will succeed. Our aspiration should thus be to *constrain* such an adversary by detecting when attacks move the system outside of a state we can trust. Rowe [59, 60] demonstrates a formal argument for instrumenting a system to force a more difficult attack; placing a higher burden on the adversary to corrupt a system and go undetected. This suggests a more practical notion of system-level security where we isolate trusted components and understand how their presence on a system confines the adversary. From this perspective the primary utility of formally verified attestation components is their *isolation of risk* on layered systems rather than guarantees of absolute security.

Deconstructing a system's security into axioms about trusted components moves the threat of system compromise to components that are much less likely to fail. Security researchers understood early on that failure of complex systems was inevitable [40] and advocated for development of core components that are

most fundamental to system security. A number of such components have come to maturity in recent years with the help of formal verification and mechanized proof. An incomplete list of high-assurance tools relevant to system-level security appear in Section 9.2. Components like these are not only critical for the development of the sound attestation infrastructure proposed herein, but also as services to be invoked and coordinated by the same infrastructure.

1.3 Overview of Contributions

In each of the attestation scenarios outlined in Section 1.1 above, the appraiser does not have direct access to the target of measurement and must rely on indirect evidence by trusted observers. Implicit in these target architectures is a high-privilege thread of control responsible for invoking attestation services and bundling results. This component is called an *attestation manager*, and is responsible for carrying out a sequence of actions on the target platform as a part of an *attestation protocol* [12]. An attestation manager must be aware of all attestation capabilities on its platform and faithfully invoke them as specified by the appraiser's request. It must also bundle the evidence such that it is cryptographically sound and amenable to checking by the appraiser. Correct execution of the attestation manager is critical to any attestation infrastructure, and thus critical to any trust an appraiser may gain in the target platform.

The aim of the current work is to design, implement, and prove correct a collection of software components that provide a sound infrastructure for remote attestation of layered systems. *Correct* means that the components faithfully carry out the goals of attestation, and *prove* means formal verification by machine-checked proof. There are many modern success stories of formal verification (See

Section 9.2 for an overview), and it has become clear that its "trustworthiness...far surpasses the confidence levels we rely on in engineering or mathematics for our daily survival" [33]. There is ongoing work to incorporate formal methods as an alternative to exhaustive testing and manual review in achieving high levels of certification for critical components in avionics and other industries where failure of components is catastrophic [21, 26, 44, 61]. Despite its advantages, formal verification remains a difficult task requiring specialized expertise and tools. Because remote attestation is critical to system-level security of modern computing environments with diverse topologies, it is worth this effort to achieve the highest levels of trust in core components that support it. The remainder of this introduction provides an overview of each contribution, followed by tables with related publications and links to code artifacts.

Copland Language and Reference Semantics

protocols; a shared vocabulary to communicate attestation goals. Copland [56] is a domain-specific language and formal framework that supports specification and analysis of layered attestation protocols. The design of Copland was joint work with myself, my advisor, and our collaborators at MITRE, John's Hopkins APL, and the NSA. The Copland language consists of attestation terms called phrases and evidence terms that describe the cryptographic structure of attestation results. Phrases intentionally leave primitive attestation services abstract, but capture their precise ordering within a protocol and also the layered relationships among protocol participants. Accompanying the language specification is a collection of reference semantics: denotational semantics for measurement ordering and evidence shapes, and an operational semantics for event traces. We will

see later how the denotational semantics support higher-level analysis, while the operational semantics aids in refinement to a more executable attestation semantics. Chapter 2 provides an introduction to Copland by example. Chapter 3 gives the full language definition and an overview of the reference semantics.

Execution Semantics: Attestation and Appraisal

While Copland provides a vocabulary for attestation goals and a foundation for comparing protocol alternatives, its reference semantics leave many details of attestation underspecified. The Copland Compiler and Copland Virtual Machine (CVM) cooperate to define a fine-grained notion of execution with explicit invocation of services and bundling of raw evidence results. The compiler translates a Copland phrase into a sequence of commands to be executed in the CVM. These commands manage interactions with remote and local-parallel attestation domains. A dual generalized appraisal procedure unbundles the raw evidence segments produced by arbitrary Copland-based executions within the CVM. All components are implemented as functional programs in the Coq [66] proof assistant, and are freely available on GitHub [50]. Full descriptions of the CVM and generalized appraisal procedure appear in Sections 4.1 and 4.2, respectively.

Verification

The Copland Virtual Machine and its dual generalized appraisal procedure provide a principled way to carry out Copland attestations and evaluate evidence results. Both components invoke external services with precise parameters and handle raw evidence bundles with intricate underlying cryptographic structure. The subtle and fine-grained nature of these components and their interaction, com-

bined with their critical role in system-level security, warrants formal verification. Because these components are defined within the Coq interactive theorem prover, we can reason about them as first class entities. In Section 5.1 we prove correctness of CVM execution, namely that it refines the Copland reference semantics with respect to attestation event trace orderings and cryptographic evidence shape. These proofs hold for arbitrary Copland phrases, and are aided by structural lemmas for traces and domain-specific automation that targets commands in the monadic Copland Compiler. Section 5.2 defines correctness of the appraisal procedure in terms of *appraisal coverage*, namely that all measurements are checked for proper contents and cryptographic integrity according to the Copland-based request term. This verification uncovered corner cases where certain evidence shapes elude appraisal, thus leading to an alternative formulation of the coverage theorem that restricts evidence shapes supported by the CVM.

Appraisal Soundness

Even with strong correctness properties like CVM event ordering and appraisal coverage, it remains unclear what one can safely infer about a target system based on a given appraisal result. The strength of such inferences is at the core of a property we call *appraisal soundness*. Our motivation for discussing appraisal soundness is not to arrive at a formal theorem that declares once-and-for-all that attestation implies absolute security. Instead we aim for a more nuanced notion that incorporates a broader view of attestation contexts. In Chapter 6 we position appraisal soundness within a novel workflow called the Copland Verification Architecture. This workflow leverages the formal attestation and appraisal components, along with an accompanying security architecture, to discharge as-

sumptions within an existing higher-level model finder. This model finder analyzes Copland attestations with respect to an active adversary attempting to subvert attestation. To exercise this framework, in Section 6.4 we perform an iterative design and analysis of a Copland attestation scenario for a UAV/Ground Station demonstration platform for the DARPA C.A.S.E. program.

Instantiating Flexible Mechanisms

The Copland Virtual Machine takes an attestation protocol specified in Copland along with initial evidence, and produces an evidence bundle that is input-compatible with the generalized appraisal procedure. The Copland Verification Architecture lifts the formal properties of these artifacts into a higher-level analysis framework to characterize an active adversary's ability to thwart a given attestation goal. While the UAV demonstration platform exercised these formal components to gain confidence in a singular attestation *design*, it left unresolved how an implementation might incorporate the formal components as *executable artifacts*, together with non-formal components, to support a diverse collection of attestation patterns. In Chapter 7 we introduce the Haskell Attestation Manager prototype implementation and demonstrate how it supports a larger collection of attestation shapes, namely the *Flexible Mechanisms* attestation scenarios proposed in the work of Helble et. al. [27]. These attestation patterns serve as an evolving benchmark that any attestation framework should aim to support.

1.4 Publications and Code Artifacts

Publications	
Artifact	Publication
Copland language and semantics	POST 2019 [56]
Copland Interpreter prototype	HotSoS 2019 [51]
Copland Compiler + CVM Verification	NFM 2021 [52]
Appraisal Soundness (DARPA UAV case study)	MEMOCODE 2021 [53]
CVM (extended) + Appraisal Verification	ISSE Journal (accepted, pending final review)

Code Artifacts	
Artifact	Link
Copland Reference Semantics	`https://ku-sldg.github.io/copland/resources/coplandcoq/index.html`
CVM + Appraisal Verification	`https://github.com/ku-sldg/copland-avm`
Haskell AM Prototype	`https://github.com/ku-sldg/haskell-am`
CakeML AM Prototype	`https://github.com/ku-sldg/am-cakeml`
DARPA UAV Models	`https://github.com/ampetz/memocode21_models`
Copland JSON Schemas	`https://github.com/ku-sldg/json-am`

Chapter 2

Copland by Example: Virus Checking as Attestation

A simple motivating example for Copland is treating virus checking as attestation. Suppose that an appraiser would like to establish if a target system is virus free. The obvious approach is for the appraiser to request virus checking results as an attestation of the remote machine and appraise the result to determine the remote machine's state. An initial specification of this attestation scenario in Copland is:

$$@_{\mathsf{p}}\,(\mathsf{vc}\ \mathsf{p}\ \mathsf{t})$$

asking platform p to invoke its own virus checker vc as an *attestation service provider* (ASP). The additional parameters to the ASP tell vc to target applications t also running at p.

Simply doing a remote procedure call places full trust in vc and its operational environment. The target could lie about its results or an adversary could tamper with the virus checking system by compromising the checker or its signature file.

An adversary could also compromise the operational environment running the checker or execute a man-in-the-middle replay attack.

A stronger attestation makes a request of the target that includes a nonce to ensure measurement freshness. The target could acquire the nonce, gather evidence from the checker, and bundle it with the nonce, signing both with its private key. The appraiser would check the signature and nonce as well as the virus checker results. While the virus checker attests system state, the signature and nonce produce cryptographic *meta-evidence* describing how evidence is handled. The Copland phrase for this attestation is:

$$*\mathsf{app}, n : @_{\mathsf{p}} [\ (\mathsf{vc\ p\ t}) \rightarrow \mathsf{SIG}\ ]$$

adding an input nonce, n, and asking p to sign the measurement result. Here the notation $*\mathsf{app}, n :$ indicates that the appraiser place app is the top-level entity in the protocol responsible for generating the nonce and appraising the final evidence.

Evidence from the virus checker may still be compromised if the virus checker executable or signature file were compromised by an adversary. The attestation protocol can be improved to return a measurement of the checker's operational environment in addition to virus checking results. The Copland phrase for this stronger attestation is:

$$\begin{aligned} &*\mathsf{app}, n : \\ &\quad @_{\mathsf{p}} [\ @_{\mathsf{ma}} [(\mathsf{attest\ p\ sys}) \rightarrow \mathsf{SIG}] \ \rightarrow \\ &\qquad (\mathsf{vc\ p\ t}) \rightarrow \mathsf{SIG}\ ] \end{aligned}$$

where ma is a trusted and isolated *measurement and attestation* domain with read access to p's execution environment. We refer to such a domain that orchestrates Copland primitives and bundles evidence results as an *attestation manager*. While

abstract for now, the attest ASP's role is as a composite measurement of sys, the virus checking infrastructure–p's operating system along with the virus checker executable and signature file.

Measurement order is critical. An active adversary may compromise a component, engage in malice, and cover its tracks while avoiding detection. Ordering constrains the adversary by making this process more difficult [60]. If the virus checker is run before its executable or signature file are hashed the adversary has much longer to compromise the checker than if they are hashed immediately before invoking the checker. Ensuring measurement order is thus critical when verifying attestation protocols and critical to any execution or transformation of protocol representations.

The attestation becomes yet stronger by extending to include the signature file *server* used to update application signatures. This server operates on a different system that is remote to the system being appraised. However, its state impacts the overall state of the virus checking infrastructure. The target system can include information about the server by performing a *layered attestation* where evidence describing the remote signature server is included in the target's evidence. Here the target p forwards an attestation request to the signature file server sf that responds in the same manner as p:

$$
\begin{aligned}
&*\mathsf{app}, n : \\
&\quad @_{\mathsf{p}}[\ @_{\mathsf{sf}}[(\mathsf{attest\ sf\ server}) \rightarrow \mathsf{SIG}] \ \rightarrow \\
&\qquad @_{\mathsf{ma}}\,[(\mathsf{attest\ p\ sys}) \rightarrow \mathsf{SIG}] \quad \rightarrow \\
&\qquad (\mathsf{vc\ p\ t}) \rightarrow \mathsf{SIG}\ ]
\end{aligned}
$$

Although this new phrase is stronger, it adds assumptions about the trust-

worthiness of the involved participants. Notice that the attestation mechanism at the signature file domain sf must be trusted to perform an attestation of its own server. This is in contrast to the cross-domain attestation performed by the ma domain that observes sys on the target place p. This level of trust in sf may be warranted if it is a known good entity, or if trust is established by other means (boot-time measurements, prior dynamic attestations, etc.).

In addition to trustworthiness, this new phrase adds assumptions about the capabilities and configurability of the participants. In particular, the top-level appraiser place app must be preconfigured with golden values and cryptographic materials necessary to evaluate the entire evidence package. While feasible, this places a high burden on the appraiser to interpret evidence produced by the signature file server, ma domain, and virus checker, collectively. It also risks violating privacy concerns of the attesting parties that may wish to constrain disclosure [12] of their system states.

As a final alternative to exercise these trade-offs, consider the following phrase:

$$
\begin{aligned}
&*\mathsf{rp}, n : \\
&\quad @_{\mathsf{p}}[\ @_{\mathsf{ma}}\,[(\mathsf{attest\ p\ sys}) \rightarrow \mathsf{SIG}]\ \rightarrow \\
&\qquad (\mathsf{vc\ p\ t}) \rightarrow \mathsf{SIG}\ \rightarrow \\
&\qquad @_{\mathsf{app}}[(\mathsf{appraise\ app}\ (\mathsf{p}, \mathsf{sys}, \mathsf{vc})) \rightarrow \mathsf{SIG}]\ ]
\end{aligned}
$$

Here the attestation of the signature file server is replaced by an explicit *appraisal* ASP performed at the end of the protocol. Notice the top-level place has also changed to rp (relying party); the client attempting to make a trust decision about the target platform p. With this new phrase, rp can delegate evaluation of the evidence to a specialist appraiser at place app. This appraiser also acts

as a a trusted-third-party between mutually-distrustful targets and clients. As we will see in Section 4.2, the formal semantics of Copland supports a general strategy for implementing such an appraisal that leverages the precise structure of the accumulated evidence bundle.

While the virus checking-as-attestation example is trivial on the surface, it exposes critical characteristics of attestation protocols that motivate and impact verification:

- Flexible mechanism—There is no single way for performing attestation or appraisal. A language-based approach for specifying attestation protocols is warranted [12].
- Order is important—Confidence in measurement ordering is critical to trusting an appraisal result. Preserving measurement ordering from protocol specification to execution is a critical correctness property [56, 59, 60].
- Trust is relative—Different attestations and appraisals result in different levels of trust. Formally specifying the semantics of attestation and appraisal is necessary for choosing the best protocol [12, 13].

Chapter 3

Copland

Copland is a domain-specific language and formal semantics for specifying remote attestation protocols. A *Copland phrase* is a term that specifies the order and place where an attestation manager invokes primitive attestation services. Copland is designed with expressivity and generality as foremost goals. As such the language parameterizes attestation scenarios over work leaving specifics of measurement, cryptographic functions, and communication capabilities to protocol negotiation and instantiation. While the definitive features and design goals of Copland are documented separately [56], we proceed with an overview due to the central role it plays in the attestation components presented herein.

3.1 Copland Phrases

The Copland phrase grammar appears in Figure 3.1. The non-terminal A represents primitive attestation actions including measurements and evidence operations. The constructor ASP defines an Attestation Service Provider atomic measurement primitive with four static parameters (abbreviated by $\bar{a}$): m, $\bar{s}$, p,

and r that identify the measurement, an optional list of string parameters, the place where the measurement runs, and the measurement target, respectively. A *place* parameter identifies a distinct attestation environment, and supports cross-domain measurements that chain trust across attestation boundaries. Since parameters to an ASP are static, they are populated during the negotiation phase of a protocol, and it is up to the protocol participants to ensure they are properly supported by the platforms involved. Note that the ASP constructor subsumes the USM and KIM term functionality from the original Copland design [56].

$$\begin{array}{rcl} t & \leftarrow & A \mid @_p\, t \mid (t \rightarrow t) \mid (t \overset{\pi}{\prec} t) \mid (t \overset{\pi}{\sim} t) \\ A & \leftarrow & \mathsf{ASP}\ \bar{a} \mid \mathsf{CPY} \mid \mathsf{SIG} \mid \mathsf{HSH} \end{array}$$

Figure 3.1. Copland Phrase grammar where:
$\bar{a} = (m,\ \bar{s},\ p,\ r)$; $m = asp\_id \in \mathbb{N}$; $\bar{s}$ is a list of string arguments; $p = place\_id \in \mathbb{N}$; $r = target\_id \in \mathbb{N}$; and $\pi = (\pi_1, \pi_2)$ is a pair of evidence splitting functions.

Remaining primitive terms specify cryptographic operations over evidence already collected in a protocol run. CPY, SIG, and HSH copy, sign and hash evidence. Although seemingly benign, when combined with other cryptographic operations and the data branching operators described later, CPY makes Copland strictly more expressive. The cryptographic implementations underlying SIG and HSH, along with the parameters of an ASP, are static and bound during protocol selection by a subset of the protocol participants. Such a negotiation should also ensure proper configuration of the platforms involved. SIG, and HSH rely implicitly on the current evidence as input and cryptographically transform that evidence.

The key to supporting attestation of layered architectures is the remote request operator, @, that allows attestation managers to request attestations on behalf

of each other. The subscript p specifies the place to send the attestation request and the subterm t specifies the Copland phrase to send. As an example, the phrase $@_1(@_2(\mathsf{t}))$ specifies that the attestation manager at place 1 should send a request to the attestation manager at place 2 to execute the phrase t. Nesting of @ terms is arbitrary within a phrase allowing expressive layered specifications parameterized over the attestation environment where they execute.

The three structural Copland terms specify the order of execution and the routing of evidence among their subterms. The semantics of these operators are diagrammed in Figures 3.2, 3.3, and 3.4 below. The phrase $\mathsf{t1} \rightarrow \mathsf{t2}$ specifies that t1 should finish executing strictly before t2 begins with evidence from t1 consumed directly by t2. This "before" relation is captured by the dotted line and $<$ symbol between terms in the diagrams below. The phrase $\mathsf{t1} \overset{\pi}{\prec} \mathsf{t2}$ specifies that t1 and t2 run in sequence with π specifying how input evidence is routed to the subterms. Conversely, $\mathsf{t1} \overset{\pi}{\sim} \mathsf{t2}$ places no restriction on the order of execution for its subterms allowing parallel execution. Both branching operators ($\prec$ and $\sim$) produce the product of executing their subterms.

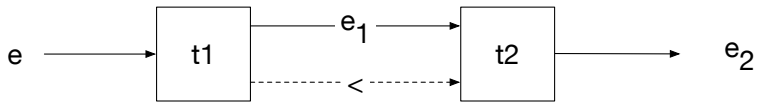

Figure 3.2. Diagram for the Copland phrase $t1 \rightarrow t2$.

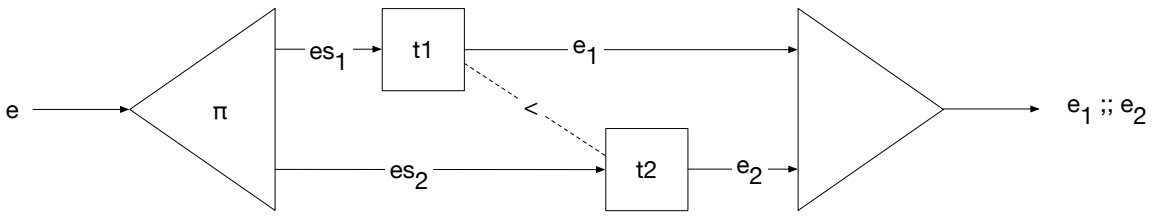

Figure 3.3. Diagram for the Copland phrase $t1 \overset{\pi}{\prec} t2$.

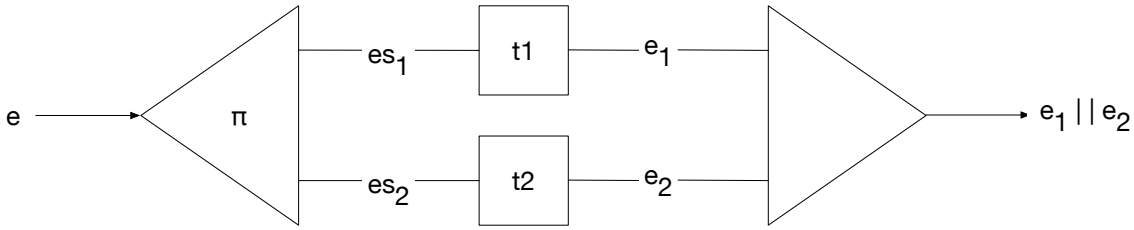

Figure 3.4. Diagram for the Copland phrase $t1 \overset{\pi}{\sim} t2$.

3.2 Copland Evidence Types

The original Copland specification [56] introduces a structured datatype to represent results of executing Copland-based attestation protocols. While referred to simply as *evidence* throughout that work, in what follows we distinguish it from other evidence representations by calling it an *Evidence Type*. Figure 3.5 shows the Evidence Type grammar, defined recursively over the non-terminal E_T. This is largely a restating of the evidence structure from Ramsdell et. al. [56] with minor syntactic tweaks. Each constructor of E_T represents a different class of evidence collected by a Copland protocol and its parameters capture the context where such measurements occurred.

$$
\begin{aligned}
E_T \;\leftarrow\; & \mathsf{mt} \mid \mathsf{N_E}\ \mathsf{n} \mid \mathsf{ASP_E}\ \bar{a}\ p\ E_T \\
& \mid \mathsf{SIG_E}\ p\ E_T \mid \mathsf{HSH_E}\ p\ E_T \\
& \mid \mathsf{SS_E}\ E_T\ E_T \mid \mathsf{PP_E}\ E_T\ E_T
\end{aligned}
$$

Figure 3.5. Evidence Type grammar where:
$\bar{a}$ and p are as in Fig. 3.1 and $n = \mathit{nonce\_id} \in \mathbb{N}$.

As an example, the $\mathsf{ASP_E}$ constructor describes evidence collected by a primitive ASP, and its parameters correspond to those of the originating ASP phrase plus a tag indicating the *measuring place*. The recursive E_T parameter indicates a nested evidence structure holding results collected earlier in the protocol. The

more deeply nested the evidence, the earlier it was collected during protocol execution. However this remains merely a *convention* of evidence construction: measurement ordering is enforced by the Copland execution semantics alone.

$\mathsf{SIG_E}$ and $\mathsf{HSH_E}$ represent digital signature and hash evidence, and each take a parameter p to tag the place performing the operation and a recursive parameter E_T for the evidence type of the payload. $\mathsf{N_E}$ represents nonce evidence, and is unique among Evidence Types because it does not have a corresponding Copland phrase originator–nonces are generated before Copland phrase execution begins and embedded as initial evidence. Finally, $\mathsf{SS_E}$ and $\mathsf{PP_E}$ represent compound evidence collected sequentially and in parallel, respectively.

3.3 Copland Reference Semantics

A critical property to consider when analyzing layered attestations is the order in which system measurement events occur. A security decision derived from attestation often hinges on a strong guarantee that one component was measured before another, or measured before it measures other components. In addition to measurement routines, an attestation infrastructure must manage components that reliably bundle and route the resulting evidence. Prior work implementing prototype attestation systems highlighted the tedious nature of these components and their interaction [51]. In the rest of this section we will give an overview of a collection of denotational and operational semantics for Copland executions that characterize attestation event traces, their ordering, and cryptographic evidence shapes. Together these formalisms provide a foundation for higher-level analysis of attestation scenarios, while also acting as a reference semantics to constrain the behavior of components that refine the behavior of Copland executions.

3.3.1 Copland Evidence Semantics

A helpful way to view an Evidence Type is as an *evidence shape*: it captures everything about measurement results and their structure, but omits concrete binary data values. Because this shape is independent of dynamic measurement results, we can precompute an expected Evidence Type with the evidence denotation function $\mathcal{E}(t, p, e)$ (originally defined in Ramsdell et. al. [56]) which takes as input a Copland phrase t, shape of the initial evidence e, and the top-level place p. The definition of $\mathcal{E}$ is repeated below in Figure 3.6 for the sake of self-containment. Computing expected evidence shapes in this way is an important prerequisite for the generalized appraisal procedure introduced later in Section 4.2.

$$
\begin{aligned}
\mathcal{E}(\mathsf{CPY}, p, e) &= e \\
\mathcal{E}((\mathsf{ASP}\ \bar{a}), p, e) &= \mathsf{ASP_E}\ \bar{a}\ p\ e \\
\mathcal{E}(\mathsf{SIG}, p, e) &= \mathsf{SIG_E}\ p\ e \\
\mathcal{E}(\mathsf{HSH}, p, e) &= \mathsf{HSH_E}\ p\ e \\
\mathcal{E}(@_q\, t, p, e) &= \mathcal{E}(t, q, e) \\
\mathcal{E}(t_1 \rightarrow t_2, p, e) &= \mathcal{E}(t_2, p, \mathcal{E}(t_1, p, e)) \\
\mathcal{E}(t_1 \overset{\pi}{\prec} t_2, p, e) &= \mathsf{SS_E}\ e1\ e2 \\
\mathcal{E}(t_1 \overset{\pi}{\sim} t_2, p, e) &= \mathsf{PP_E}\ e1\ e2
\end{aligned}
$$

Figure 3.6. Copland Evidence Semantics.
In $\mathsf{SS_E}$ and $\mathsf{PP_E}$ cases, $\pi = (\pi_1, \pi_2)$, $e1 = \mathcal{E}(t_1, p, \pi_1(e))$, $e2 = \mathcal{E}(t_2, p, \pi_2(e))$

3.3.2 Copland Events

The first step towards formal analysis of attestation systems is to define a precise mathematical model of attestation-relevant system events. Events capture actions taken by an attestation manager, and in Copland they appear both as labels on steps in the LTS semantics of Section 3.3.3 and as leaf nodes in the

Event System partial ordering of Section 3.3.4. They fall into four categories:

Measurement: Invocation of black-box ASP routines.

Cryptographic: Copy, sign, and hash operations over prior evidence.

Communication: Request/Reply sessions for interaction with remote domains.

Evidence routing: Local split and join operations that handle evidence routing for the branching Copland phrases ($\overset{\pi}{\prec}$ and $\overset{\pi}{\sim}$).

The Copland event grammar appears in Figure 3.7. Of note is one minor difference compared to Copland's original definition of events: we omit the output Evidence Type parameter for primitive terms. This made verification more tedious without an obvious benefit to prove facts about evidence for this work.

$$
\begin{array}{rcl}
V & \leftarrow & \mathsf{ASP}_{\mathsf{event}}(\mathbb{N}, p, \bar{a}, E_T) \mid \mathsf{CPY}(\mathbb{N}, p) \\
& \mid & \mathsf{SIG}_{\mathsf{event}}(\mathbb{N}, p, E_T) \mid \mathsf{HSH}_{\mathsf{event}}(\mathbb{N}, p, E_T) \\
& \mid & \mathsf{REQ}(\mathbb{N}, p, p, t, E_T) \mid \mathsf{RPY}(\mathbb{N}, p, p, E_T) \\
& \mid & \mathsf{SPLIT}(\mathbb{N}, p) \mid \mathsf{JOIN}(\mathbb{N}, p)
\end{array}
$$

Figure 3.7. Copland Events.

Events capture observable actions performed during attestation. Actions that invoke IO may rely on external components, each with their own implementation and independent evaluation criteria. Each event is labeled by a unique natural number identifier. This label is vital during formal analysis to distinguish different *instances* of an event over time, within execution of a single protocol. Each event also has a place identifier to record the platform where the event occurred.

Bookkeeping of event and place identifiers is a verification artifact and irrelevant in a concrete implementation. Measurement and cryptographic events

correspond exactly to primitive Copland terms, where the input evidence type is captured as a parameter. Communication events REQ and RPY model a request and reply interaction to a remote protocol participant. The two place parameters capture the sender and recipient place (in that order). Evidence routing events SPLIT and JOIN record local splitting and joining of evidence that arise from the branching phrases ($\overset{\pi}{\prec}$ and $\overset{\pi}{\sim}$) that route evidence among their subterms.

3.3.3 Copland LTS Semantics

The Copland framework provides an abstract specification of Copland phrase execution in the form of a small-step operational Labeled Transition System (LTS) semantics. States of the LTS correspond to protocol execution states, and its inference rules transform a Copland phrase from a protocol description to an event trace. A single step is specified as $s_1 \overset{\ell}{\rightsquigarrow} s_2$ where s_1 and s_2 are states and ℓ is a label that records attestation-relevant system events. The reflexive, transitive closure, $s_1 \overset{c}{\rightsquigarrow}^* s_2$, collects a trace c of all such observable attestation steps. $\mathcal{C}(t, p, e)$ represents an initial configuration state with Copland phrase t, starting place p, and initial evidence e. $\mathcal{D}(p, e')$ represents the end of execution at place p with final evidence e'. Therefore, $\mathcal{C}(t, p, e) \overset{c}{\rightsquigarrow}^* \mathcal{D}(p, e')$ captures the complete execution of Copland phrase t that exhibits event trace c.

To demonstrate how the LTS semantics generates event labels we describe two representative sets of inference rules; one for the primitive term SIG and one for the compound term $\mathsf{t1} \rightarrow \mathsf{t2}$. The rule for SIG appears in Figure 3.8. Here the single step relation moves from the initial state holding a SIG term directly to the done state D. The evidence e is transformed into signed evidence $\mathsf{SIG_E}\ p\ e$, and the label v holds a SIG event. The rules for $\mathsf{t1} \rightarrow \mathsf{t2}$ in Figure 3.9 are only slightly

more involved. The first rule says that an initial configuration state holding a phrase $\mathsf{t1} \rightarrow \mathsf{t2}$ can step to the linear sequence state LS with a silent label τ. The LS state holds two parameters, where the first is another configuration state and the second is a Copland phrase. This is how the LS state *focuses* $\mathsf{t1}$ for execution. The middle rule lifts internal steps to the LS level. Once the internal state reaches the done state D, the final rule allows a silent step from the LS state to a new configuration state C that effectively focuses the $\mathsf{t2}$ term for execution. All Copland terms that appear in the LTS semantics are annotated with a range of natural numbers to facilitate the unique identifiers of event labels. The SIG term in Figure 3.8 is annotated with the trivial range $(i, i+1)$, as are all primitive terms. The SIG event grabs i from the lower bound to build its unique label.

$$\mathcal{C}([\mathsf{SIG}]^{i}_{i+1}, p, e) \overset{v}{\rightsquigarrow} \mathcal{D}(p, \mathsf{SIG_E}\ p\ e) \quad [v = \mathsf{SIG_{event}}(i, p, e)]$$

Figure 3.8. LTS rule for SIG.

$$\begin{aligned}
\mathcal{C}([\mathsf{t1} \rightarrow \mathsf{t2}]^{i}_{j}, p, e) &\overset{\tau}{\rightsquigarrow} \mathcal{LS}(\mathcal{C}(\mathsf{t1}, p, e), \mathsf{t2}) \\
\mathcal{LS}(s_1, \mathsf{t2}) &\overset{v}{\rightsquigarrow} \mathcal{LS}(s_2, \mathsf{t2}) \qquad \text{if } s_1 \overset{v}{\rightsquigarrow} s_2 \\
\mathcal{LS}(\mathcal{D}(p, e), \mathsf{t}) &\overset{\tau}{\rightsquigarrow} \mathcal{C}(\mathsf{t}, p, e)
\end{aligned}$$

Figure 3.9. LTS rules for $\rightarrow$.

3.3.4 Copland Event Systems

In addition to the operational LTS semantics, the Copland specification defines a strict partial order on attestation events called an *Event System*. Event Systems are constructed inductively where: (i) Leaf nodes represent base cases and hold a

single event instance; and (ii) Before nodes ($t1 \triangleright t2$) and Merge nodes ($t1 \bowtie t2$) are defined inductively over terms. Before nodes impose ordering while Merge nodes capture parallel event interleaving where orderings within each sub-term are maintained. The Event System denotation function, $\mathcal{V}$, maps an annotated Copland term (Copland phrases extended with a range of unique identifiers), place, and initial evidence to a corresponding Event System. A representative subset of this semantics [56] appears in Figure 3.10.

$$
\begin{array}{rcl}
\mathcal{V}([\mathsf{ASP}\ \bar{a}]^i_{i+1}, p, e) & = & \mathsf{ASP}_{\mathsf{event}}(i, p, \bar{a}, \mathsf{ASP}_{\mathsf{E}}\ \bar{a}\ p\ e) \\
\mathcal{V}([\mathsf{SIG}]^i_{i+1}, p, e) & = & \mathsf{SIG}_{\mathsf{event}}(i, p, \mathsf{SIG}_{\mathsf{E}}\ p\ e) \\
\mathcal{V}([@_q\ t]^i_j, p, e) & = & \mathsf{REQ}(i, p, q, t, e) \triangleright \\
 & & \mathcal{V}(t, q, e) \triangleright \\
 & & \mathsf{RPY}(j-1, p, q, \mathcal{E}(t, q, e)) \\
\mathcal{V}([t_1\ \overset{(\pi_1,\pi_2)}{\sim}\ t_2]^i_j, p, e) & = & \mathsf{SPLIT}(i, ...) \triangleright \\
 & & (\mathcal{V}(t_1, p, \pi_1(e)) \bowtie \mathcal{V}(t_2, p, \pi_2(e))) \triangleright \\
 & & \mathsf{JOIN}(j-1, ...)
\end{array}
$$

Figure 3.10. Event System semantics (representative subset).

3.4 Copland Correctness Theorem

Taken together, Event Systems and the LTS are useful as reference semantics to characterize attestation manager execution and denote evidence structure. While Event Systems are denotational and well-suited for higher-level analysis, the LTS provides an operational view of Copland attestations by enumerating the set of allowable execution traces. The fundamental formal result of the original specification [56] brings together these two main components of the reference semantics. It says that any event v preceding an event v' in the Event System generated by the annotated Copland phrase $\mathsf{t}_{(\mathsf{i})}$ $(\mathcal{V}(\mathsf{t}_{(\mathsf{i})}, \mathsf{p}, \mathsf{e}) : v \prec v')$ also precedes v' in traces

exhibited by the LTS semantics. This fact is repeated here as Theorem 1, where the notation $v \ll_{\mathsf{c}} v'$ means "v precedes v' in event sequence c":

Theorem 1 (LTS_Respects_Event_System)

$\mathcal{C}(\mathsf{t}_{(\mathsf{i})}, \mathsf{p}, \mathsf{et}) \overset{\mathsf{c}}{\rightsquigarrow}{}^{*} \mathcal{D}(\mathsf{p}, \mathsf{et}')$

$\wedge \quad \mathcal{V}(\mathsf{t}_{(\mathsf{i})}, \mathsf{p}, \mathsf{et}) : v \prec v' \;\Rightarrow\; v \ll_{\mathsf{c}} v'.$

Figure 3.11 shows a concise graphical summary of this theorem, relating the different semantic notions of Copland phrase execution. While this result is critical for linking the LTS and Event System semantics, the LTS remains an abstract *relational*, non-executable artifact. A contribution of the current work is to refine the LTS semantics to a more fine-grained notion of Copland execution that performs explicit invocation of services and bundling of raw evidence. By linking this refinement to the existing LTS specification, we can leverage Theorem 1 to ensure that analysis over Event Systems is sound with respect to concrete executions. Section 4.1 introduces the fine-grained semantics for Copland execution, and we make its connection to Theorem 1 explicit in Section 5.1.

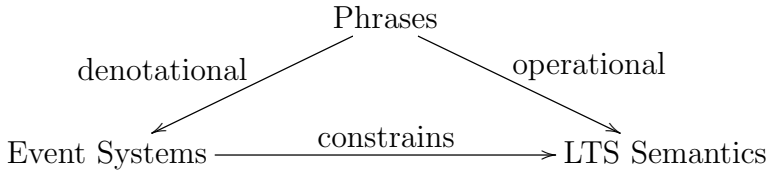

Figure 3.11. Semantic Relations. (Figure 1 from Section 1 in [56]).

Chapter 4

Execution Semantics: Attestation and Appraisal

While the LTS and Event System semantics give abstract characterizations of Copland attestations, we now move to a more fine-grained notion of execution with explicit invocation of services and bundling of evidence. The *Copland Compiler* translates a Copland phrase into a sequence of commands to be executed in the *Copland Virtual Machine* (CVM), a monadic run-time that orchestrates attestation primitives. A dual *generalized appraisal procedure* unbundles raw evidence produced by the CVM and evaluates it for measurement content and cryptographic integrity. Together these components provide a principled pipeline to execute, bundle, and evaluate arbitrary Copland-based attestations. The compiler, CVM, and appraisal procedure are implemented as monadic functional programs in the Coq [66] proof assistant, and are freely available on GitHub [50].

4.1 Copland Virtual Machine

The Copland Virtual Machine Monad is a state and exception monad defined in Coq with the primitives bind, return, put, and get implemented in the canonical way. While standard, the generic monadic type St and some accompanying automation were adapted from the Verdi framework for formally verifying distributed systems [55, 68]:

```
Definition St(S A : Type) : Type :=
    S -> (option A) * S
```

The St computation takes a state parameter of type S as input, and returns a pair of an optional return value of type A and an updated state.

The CVM Monad is a specialization of St with a CVM_st structure as the state type S. CVM_st is a record datatype with fields that hold configuration data for the CVM as it executes. These fields are st_ev, st_trace, st_pl, and st_evid that maintain an evidence bundle, a trace of attestation-relevant IO events, currently-executing place, and IO event ID counter, respectively. The CVM also comes with standard functions for executing its stateful computations (runState, evalState, execState), the always-failing computation (failM), and getters/putters specialized to the CVM internal fields.

4.1.1 Raw and Type-Tagged Evidence

Whereas an Evidence Type (Figure 3.5) describes the cryptographic shape of evidence, execution of the CVM requires a *raw evidence* representation to hold the actual binary data values gleaned from dynamic measurements and cryptographic operations over that data. In Coq we assign primitive binary data values the type

BS (short for "Binary String") which we leave uninterpreted. Larger raw evidence sequences accumulated during attestation then become lists of BS values:

```
Definition BS : Set.  Admitted.


Definition RawEv := list BS.
```

Lists are a convenient abstraction for binary data sequences, and have sufficient built-in theories in Coq to prove meaningful facts about attestation and appraisal.

While an implementation of attestation need only operate over raw measurement values, it is crucial during verification of the CVM to maintain the precise structure of the underlying evidence as it is constructed and shared among protocol participants. Mistakes in how evidence is bundled could lead to a lack of cryptographic strength or disclosure of measurement results that violate privacy expectations. With that in mind, we introduce the "Type-tagged Evidence" datatype EvC which will serve as the type of the st_ev field of CVM_st:

```
Inductive EvC: Set :=
| evc: RawEv -> EvidenceT -> EvC.
```

This representation pairs the "list of bits" raw evidence together with its Evidence Type, EvidenceT. This supports a verification strategy where each operation manipulating raw evidence is accompanied by a principled update to its evidence structure.

4.1.2 Measurement and Cryptographic Primitives

Measurement primitives build computations in the CVM Monad that perform two primary functions: simulate invocation of measurement services and explic-

itly bundle the evidence results. Each primitive follows a general pattern: grab necessary inputs from the current monadic state, simulate external IO by *tagging* an event with all relevant parameters, and finally returning an updated evidence bundle that is placed back into the CVM state for further processing.

Collection of raw evidence values is necessarily *simulated* because we cannot literally invoke IO within the Coq verification environment. However, because the datatype BS is opaque we can capture and reason about parameters to "raw-bits-returning" functions in the formal environment, but leave their implementations (custom measurement routines, cryptographic algorithms) configurable by platform owners. This approach provides a natural path from verification to synthesized code that can fill in the IO stubs.

```
Definition tag_ASP
   (params :ASP_PARAMS) (mpl:Plc) : CVM BS :=
 x <- next_event_id ;;
 let bs := (do_asp params mpl x) in
   add_tracem [asp_Event x mpl params] ;;
   ret bs.

Definition invoke_ASP
   (params:ASP_PARAMS) : CVM EvC :=
 e <- get_ev ;;
 p <- get_pl ;;
 bs <- tag_ASP params p ;;
 ret (cons_uu bs e params p).
```

Figure 4.1. Example monadic measurement primitive.

For a representative example of such a primitive, invoke_ASP and its helper function tag_ASP appear in Figure 4.1. In invoke_ASP, after grabbing the current evidence and place from the CVM state, tag_ASP simulates an IO measurement

and cons_uu bundles the raw evidence result. In tag_ASP the event identifier x is generated by next_event_id, a simple counter maintained by the CVM during verification. Because next_event_id is monotonic, x is guaranteed unique per-protocol. This accounts for multiple, independent invocations of the same ASP and captures changes in a target's state over time. Notice that x serves the dual role of tagging the ASP event in the CVM trace (via add_tracem) and as an abstract representation of a binary measurement result as a parameter to the uninterpreted IO stub do_asp.

The cons_uu helper function in Figure 4.2 performs evidence bundling within the type-tagged evidence structure. For the raw evidence portion, it simply acts as a cons operation onto the existing BS list. The rest of cons_uu extends the existing Evidence Type to tag it with the ASP parameters invoked and the measuring place from which the ASP was launched. uu is the Coq constructor corresponding to the ASP_E Evidence Type.

Additional monadic primitives exist for cryptographic operations like signing and hashing, but we elide their definitions because they are structurally identical to invoke_ASP. The main difference is in their "evidence bundlers" cons_gg and cons_hh, which appear at the end of Figure 4.2. These account for the subtle differences in semantics for signing and hashing, respectively. Of note is how cons_hh overwrites its existing raw evidence with the single new hash value.

This new hash of type BS is computed as follows: do_hash (encodeEvBits e) p, where p and e represent the hashing place and current raw evidence in the CVM. do_hash and encodeEvBits are left uninterpreted to keep the hashing algorithm and binary evidence representation abstract. In contrast to cons_hh, cons_gg retains its input raw evidence payload along with the signature over that payload;

aligning with common practice for digital signatures. Nonetheless, both cons_hh and cons_gg update the Evidence Type to extend the cryptographic structure of the underlying payload.

```
Definition cons_uu
    (x:BS) (e:EvC) (params:ASP_PARAMS)
    (mpl:Plc) : EvC :=
  match e with
  | evc bits et =>
      evc (x :: bits) (uu params mpl et)
  end.

Definition cons_gg
    (sig:BS) (e:EvC) (p:Plc): EvC :=
  match e with
  | evc bits et =>
      evc (sig :: bits) (gg p et)
  end.

Definition cons_hh
    (hsh:BS) (e:EvC) (p:Plc): EvC :=
  match e with
  | evc _ et =>
     evc [hsh] (hh p et)
  end.
```

Figure 4.2. Evidence bundlers: cons_uu, cons_gg, cons_hh.

4.1.3 Remote and Parallel CVM Execution

When interpreting a remote request term $@_p t$ or a parallel branch $t1 \overset{\pi}{\sim} t2$ CVM execution relies on interaction with an external attestation manager that is also running an instance of the CVM. tag_REQ in Figure 4.3 captures an IO request and its parameters: t (Copland phrase requested), p (requesting place),

q (destination place), and e (the initial evidence forwarded to q). Capturing the Evidence Type at each request is essential to denote precisely which segments of the underlying raw evidence are shared, and with whom, during protocol execution. This in turn supports higher-level analyses that involve privacy expectations of protocol participants.

```
Definition tag_REQ (t:Term) (p:Plc) (q:Plc)
    (e:EvC) : CVM unit :=
  reqi <- next_event_id ;;
  add_tracem [req reqi p q t (get_et e)].
```

Figure 4.3. Tagging a request Event.

Whereas tag_REQ models a request to a remote CVM, remote_session (Figure 4.4) models the events and evidence generated by a CVM carrying out the request at a remote end-point. cvm_events t q et is an uninterpreted function used *only during verification* to simulate the trace of attestation events emitted by executing Copland phrase t at place q with initial evidence of type et. doRemote_session is also uninterpreted, but is an implementation-relevant configurable IO stub, akin to do_asp. Its intended semantics is an IO-blocking communication procedure that facilitates a request/response session with a remote CVM, returning the evidence received in the response. Finally, doRemote (Figure 4.4) is the top-level monadic function combining these communication primitives in sequence. Upon receipt of evidence from the remote CVM it tags the reply event and returns the evidence.

Parallel execution of Copland phrases is modeled by an asynchronous interaction with "local parallel" CVM instances. Figure 4.5 shows the two monadic prim-

```
Definition doRemote
    (t:Term) (q:Plc) (e:EvC) : CVM EvC :=
  p <- get_pl ;;
  tag_REQ t p q e ;;
  e' <- remote_session t q e ;;
  tag_RPY p q e' ;;
  ret e'.

Definition remote_session
    (t:Term) (q:Plc) (e:EvC) : CVM EvC :=
  add_tracem (cvm_events t q (get_et e)) ;;
  ret (doRemote_session t q e).
```

Figure 4.4. Monadic communication session.

itives, start_par_thread and wait_par_thread, that facilitate this interaction. Each takes as input a *thread location* of type Loc that acts as an abstract handle to outstanding CVM threads of execution. During compilation (See Section 4.1.4) the Loc parameters are derived from annotations on parallel Copland phrases to be unique per start/wait pair for a given phrase.

start_par_thread invokes the uninterpreted IO function do_start_par_thread loc t e to launch a parallel CVM thread identified by location *loc*, asking it to execute Copland phrase t with initial raw evidence e. It also adds a corresponding cvm_thread_start event to tag the launch with its relevant parameters. cvm_thread_start and cvm_thread_end are events unique to traces emitted by the CVM; they do not appear in the LTS reference semantics. They serve to bookend other events, allowing the main CVM thread to continue emitting events while its parallel threads complete. do_wait_par_thread loc t p e simulates poll-waiting for an evidence result at thread location *loc*, and as soon as evidence "arrives" at *loc*, the CVM emits a cvm_thread_end loc event and returns the evidence.

```
Definition start_par_thread (loc:Loc)
    (t:AnnoTerm) (e:EvC) : CVM unit :=
  p <- get_pl ;;
  do_start_par_thread loc t (get_bits e) ;;
  add_tracem
    [cvm_thread_start loc p t (get_et e)].

Definition wait_par_thread (loc:Loc)
    (t:AnnoTerm) (e:EvC) : CVM EvC :=
  p <- get_pl ;;
  e' <- do_wait_par_thread loc t p e ;;
  add_tracem [cvm_thread_end loc] ;;
  ret e'.
```

Figure 4.5. Parallel execution monadic primitives.

Because remote and parallel instances of the CVM are external to the main thread of execution, during verification we must make assumptions about the evidence they collect. Towards this, we define an uninterpreted function to represent a "golden evidence semantics" for CVM execution. Because the core CVM semantics should be identical for valid remote and local parallel instances, we provide rewrite rules as Axioms to equate their respective IO stub specializations. This supports sharing of proof machinery for assumptions involving external CVM instances while still maintaining the distinction between remote and local parallel IO stubs. This approach enables a smoother translation to concrete implementations where differences in their glue code may be significant.

4.1.4 Copland Compiler

Each case of the Copland Compiler in Figure 4.6 pattern matches on the syntax of an annotated Copland phrase, then uses the monadic sequence operation ;; to build a corresponding series of commands in the CVM Monad. The indi-

vidual commands are not *executed* by the compiler directly, but returned as a computation to be executed later. This approach is inspired by work that uses a monadic shallow embedding in HOL to synthesize stateful CakeML code [28]. The shallow embedding style [22] allows the protocol writer to leverage the sequential, imperative nature of monadic notation while also having access to a rigorous formal environment to analyze chunks of code written in the monad. It also leverages Coq's built-in name binding metatheory, avoiding this notoriously difficult problem in formal verification of deeply embedded languages [2].

The first three compiler cases are trivial. The ASP term case invokes the do_prim function that dispatches commands specific to each primitive Copland operation (i.e. invoke_ASP from Section 4.1.2). The @ term case invokes doRemote, bookended by evidence management. Finally, the linear sequence term $(t_1 \rightarrow t_2)$ case invokes copland_compile recursively on the subterms t_1 and t_2 and appends the results in sequence. Note that because evidence is cumulative in the CVM, this means that evidence computed by executing the t_1 commands will serve as input to the t_2 commands.

The branch sequence case $(t_1 \overset{(sp1,sp2)}{\prec} t_2)$ filters the initial evidence into evidence for the two subterms using the split_ev helper. The commands for the t_1 and t_2 subterms are then compiled in sequence, placing initial evidence for the respective subterm in the CVM_st before executing each, and extracting evidence results after. join_seq combines result evidence and emits a join event.

The branch parallel case $(t_1 \overset{(sp1,sp2)}{\sim} t_2)$ begins like the branch sequence case, using split_ev to split the initial evidence. But now we instead invoke the start_par_thread monadic helper function to launch execution of the t_2 sub-commands in a parallel CVM instance thread. We then proceed to compile the t_1 subterm as usual in

the main CVM thread and extract its evidence result from the CVM_st. This case concludes by invoking wait_par_thread to poll-wait for the evidence result from the $t2$ thread, then packaging evidence results and emitting a join event via join_par. We note here that our decision to leave the thread model abstract in the CVM semantics allows for attestation managers to run in diverse environments that may or may not provide native support for concurrency.

Monadic values represent computations waiting to run. run_cvm t st (end of Figure 4.6) interprets the monadic computation (copland_compile t) with initial state st, producing an updated state. This updated state contains the collected evidence and event trace corresponding to execution of the input term and initial evidence in st. The evidence and event trace are sufficient to verify correctness of run_cvm with respect to the LTS semantics.

4.2 Appraisal

Appraisal is the final step in a remote attestation protocol where an indirect observer of a target platform must analyze evidence in order to determine the target's trustworthiness. Regardless of its level of scrutiny, an appraiser must have a precise understanding of the structure of evidence it examines. The Copland framework provides such a shared evidence structure, and Copland phrases executed by the CVM produce evidence with a predictable shape. In this section we introduce a generalized strategy for appraisal of raw evidence that leverages Copland's evidence semantics along with an environment for accessing golden measurement values and cryptographic materials.

```
Fixpoint copland_compile
    (t:AnnoTermPar): CVM unit :=
  match t with
  | aasp_par a =>
      e <- do_prim a ;;
      put_ev e
  | aatt_par q t' =>
    e <- get_ev ;;
    e' <- doRemote t' q e ;;
    put_ev e'
  | alseq_par t1 t2 =>
      copland_compile t1 ;;
      copland_compile t2
  | abseq_par sp t1 t2 =>
    (e1,e2) <- split_ev sp ;;
    put_ev e1 ;;
    copland_compile t1 ;;
    e1r <- get_ev ;;
    put_ev e2 ;;
    copland_compile t2 ;;
    e2r <- get_ev ;;
    join_seq e1r e2r
  | abpar_par loc sp t1 t2 =>
    (e1,e2) <- split_ev sp ;;
    start_par_thread loc t2 e2 ;;
    put_ev e1 ;;
    copland_compile t1 ;;
    e1r <- get_ev ;;
    e2r <- wait_par_thread loc t2 e2 ;;
    join_par e1r e2r
  end.

Definition run_cvm
    (t:AnnoTermPar) st : cvm_st :=
  execSt (copland_compile t) st.
```

Figure 4.6. The Copland Compiler–builds computations as sequenced CVM commands.

4.2.1 Typed Concrete Evidence

While the type-tagged evidence representation introduced earlier provides a clean separation between raw evidence and its structure during CVM execution, for appraisal it is convenient to merge the two into a single *Typed Concrete Evidence* type shown in Figure 4.7. E_{Tc} is similar in structure to the Evidence Type grammar of Copland (E_T), but with raw binary evidence values bs inserted inline. One exception is the hash evidence constructor $\mathsf{HSH_c}$ whose recursive parameter is an Evidence *Type* E_T rather than concrete evidence: the one-way hash operation "forgets" the input raw evidence but still maintains its underlying structure for use during appraisal.

$$
\begin{array}{rcl}
E_{Tc} & \leftarrow & \mathsf{mt_c} \mid \mathsf{ASP_c}\ \bar{a}\ p\ \mathsf{bs}\ E_{Tc} \mid \mathsf{SIG_c}\ p\ \mathsf{bs}\ E_{Tc} \\
& & \mid \mathsf{HSH_c}\ p\ \mathsf{bs}\ E_T \mid \mathsf{N_c}\ \mathsf{n}\ \mathsf{bs} \mid \mathsf{SS_E}\ E_{Tc}\ E_{Tc} \\
& & \mid \mathsf{PP_E}\ E_{Tc}\ E_{Tc}
\end{array}
$$

Figure 4.7. Typed Concrete Evidence grammar where: $\bar{a}$, p, and n are as in Fig. 3.5 and $\mathsf{bs} \in \mathsf{BS}$ (binary values).

The E_{Tc} evidence representation is convenient for both appraisal verification and execution. During verification it allows a more natural "evidence subterm" relation, and during execution it is useful as an "all-in-one" structure for remote parties to communicate attestation and appraisal results, along with custom certificates over evidence. Nonetheless, the raw evidence sequence alone is vital as an intermediate form for attestation scenarios with mutual distrust where an intermediary must not know the semantic structure of the evidence it handles.

4.2.2 Generalized Appraisal Procedure

The CVM takes an arbitrary Copland phrase as input and generates an evidence bundle with predictable shape. We now aim to define a dual mechanism to perform appraisal by *deconstructing* an evidence bundle to check both its cryptographic integrity and measurement payloads against expected values. For now we assume the entity performing appraisal is the same that initiated the attestation.

Representative cases of the generalized appraisal procedure appraise appear in Figure 4.8. appraise takes as inputs an Evidence Type et and raw evidence sequence ls, and returns an optional EvidenceC (The Coq encoding of E_{Tc}). The sequence of raw values have no meaning in isolation, but gain meaningful structure when paired with a corresponding Evidence Type. In general each case of this function will begin with an attempt to peel off one or more binary values from the front of the raw evidence list. If successful, the values are passed to primitive appraisal checker functions.

The nonce and hash case (nn and hh constructors) only require the frontmost binary value to complete appraisal, while other cases require one or more recursive calls to appraise. The sequential and parallel cases (ss and pp) make recursive calls to disjoint partitions of the raw evidence list, where the partitions are computed based on the size of the left-most Evidence Type subterm. Computing partitions in this way ensures that, assuming appraise succeeds, it operates over the correct raw evidence of sufficient length.

4.2.3 Primitive Appraisal Checkers

The generalized appraisal procedure appraise in Figure 4.8 relies on primitive appraisal checker functions checkASP, checkSig, checkHash, and checkNonce.

```
Fixpoint appraise (et:EvidenceT) (ls:RawEv) :
  option EvidenceC :=
    match et with
    | nn nid =>
      (n, _) <- peel_bs ls ;;
      res <- checkNonce nid n ;;
      Some (nnc nid res)

    | uu args et' =>
      (bs, ls') <- peel_bs ls ;;
      res <- (checkASP args bs) ;;
      x <- appraise et' ls' ;;
      Some (uuc args res x)

    | gg p et' =>
      (sig, ls') <- peel_bs ls ;;
      res <- checkSig ls' p sig ;;
      x <- appraise et' ls' ;;
      Some (ggc p res x)

    | hh p et =>
      (bs, _) <- peel_bs ls ;;
      res <- checkHash et p bs ;;
      Some (hhc p res et)

    | ss et1 et2 =>
      x <- appraise et1 (firstn (et_size et1) ls) ;;
      y <- appraise et2 (skipn  (et_size et1) ls) ;;
      Some (ssc x y)

    ...

    end.
```

Figure 4.8. Generalized appraisal procedure.

They are "primitive" in the sense that their implementations are external and appraiser-configurable. Similar to the IO stubs for primitives of CVM execution, we implement appraisal checkers in Coq as mostly-uninterpreted functions; capturing precise parameters ("what" is appraised) while leaving the details of these checks abstract.

An implementation of checkASP relies on an appraiser-supplied checker function that can interpret the semantic content of the primitive ASP evidence blob. ASP appraisals range in sophistication from simple equality checks derived from a database of golden values, to more custom or qualitative analyses based on a given scenario. The checkSig primitive relies on mapping its place parameter p to the public key counterpart of the private key used to sign the evidence during attestation. Assuming the identity of the signer is linked to p, checking the signature blob ensures integrity of the raw evidence and binds it to p.

checkHash differs from checkSig in that it relies on the Evidence Type of the underlying hash blob. This is because the hash operation in the CVM reduces the already-accumulated evidence to a single value, as opposed to a signature that retains the raw evidence it signs. checkHash must *reconstruct* an expected hash value based on the Evidence Type and place parameter alone, then compare it to the actual hash value computed during attestation. Formalization of this procedure uncovered that reconstructing such a hash is problematic for certain evidence types that involve a signature embedded within a hash, and we discuss a remedy to this in Section 5.2.

Finally, checkNonce looks up a golden nonce value by ID and performs an equality check against the value in the returned evidence bundle. By design all nonce evidence is generated before a Copland attestation begins, and is embedded

as initial evidence. At generation time, each nonce is assigned a unique identifier, and a mapping from nonce ID to value is stored in an appraisal context outlined in the next section.

4.2.4 Appraisal in the AM Monad

While the CVM Monad supports faithful execution of an individual Copland phrase, many actions before and after execution are more naturally expressed at a layer above Copland. Actions preceding execution prepare initial evidence, collect evidence results from earlier runs, and generate fresh nonces. Actions following CVM execution include appraisal and preparing additional Copland phrases for execution. These pre- and post- actions are encoded as statements in the *Attestation Manager (AM) Monad*.

Rather than performing measurements directly, the AM Monad environment relies on run_cvm as a well defined interface to the CVM. This abstracts away details of Copland phrase execution and allows the consumer of an attestation to compose facts about the CVM, like those verified in Section 5.1 about events and evidence shapes, in support of higher level trust decisions. The function do_appraise in Figure 4.9 is an AM Monad computation that performs an end-to-end attestation and appraisal. Here am_appraise is a wrapper around appraise with access cryptographic materials and golden values required by primitive appraisal checkers. Finally, trust_decision is an abstract operation that takes some action based on the appraised evidence bundle.

```
Definition do_appraise (t:AnnoTerm) (p:Plc) : AM () :=
    (n, nid) <- gen_nonce ;
    bits' <- run_cvm t p n ;
    res <- am_appraise (aeval t p (nn nid)) bits' ;
    trust_decision res
```

Figure 4.9. Appraisal in the AM Monad.

Chapter 5

Verification

In Section 3.3 we defined the Copland Reference Semantics, a collection of abstract denotations that specify key behaviors of Copland attestations amenable to higher-level analysis. Later in Chapter 4 we presented a refinement of Copland executions called the Copland Virtual Machine and its accompanying generalized appraisal procedure that perform intricate orchestration, bundling, and evaluation of Copland primitives and their evidence results. However up until now there are no formal connections linking either the Reference Semantics to CVM execution, or CVM execution to appraisal. Because these artifacts are implemented as functional programs in the Coq proof assistant, we now make these connections explicit through formal specification and proof.

5.1 CVM Verification

Correctness of the Copland Virtual Machine amounts to proving that running compiled Copland terms results in evidence and event sequences that respect the Copland reference semantics. Earlier work [56] proves that any event v preceding

an event v' in an Event System generated by the annotated Copland phrase $\mathsf{t}_{(\mathsf{i})}$ ($\mathcal{V}(\mathsf{t}_{(\mathsf{i})}, p, e) : v \prec v'$) also precedes v' in the trace c exhibited by the LTS semantics $\rightsquigarrow^*$. This fact is repeated here as Theorem 2, where the notation $v \ll_{\mathsf{c}} v'$ means "v precedes v' in event sequence c". The notation $\mathsf{t}_{(\mathsf{i})}$ means Copland phrase t annotated with event ids starting at index i.

Theorem 2 (LTS_Respects_Event_System)

$\mathcal{C}(\mathsf{t}_{(\mathsf{i})}, \mathsf{p}, \mathsf{et}) \overset{\mathsf{c}}{\rightsquigarrow}{}^* \mathcal{D}(\mathsf{p}, \mathsf{et}')$

$\wedge \quad \mathcal{V}(\mathsf{t}_{(\mathsf{i})}, \mathsf{p}, \mathsf{et}) : v \prec v' \;\Rightarrow\; v \ll_{\mathsf{c}} v'.$

To verify the event semantics of the CVM we replace the LTS evaluation relation with CVM execution and show that it respects the same Event System. Theorem 3 defines this goal:

Theorem 3 (CVM_Respects_Event_System)

run_cvm *(*copland_compile t*)*

{ st_ev := (_, et), st_pl := p, st_trace := [ ], st_evid := i }

$\Downarrow$

{ st_ev := (_, _), st_pl := p, st_trace := c, st_evid := _ }

$\wedge \quad \mathcal{V}(\mathsf{t}_{(\mathsf{i})}, \mathsf{p}, \mathsf{et}) : v \prec v' \;\Rightarrow\; v \ll_{\mathsf{c}} v'.$

The $\Downarrow$ notation emphasizes that run_cvm is literally a functional program written in Coq. This differentiates it from the relational small-step LTS semantics $\overset{c}{\rightsquigarrow}{}^*$. run_cvm takes as inputs a sequence of commands in the CVM Monad and a CVM_st structure that includes fields for initial evidence (st_ev), starting place (st_pl), initial event trace (st_trace), and a starting value for the event ID counter (st_evid). It outputs the CVM_st that results from interpreting the compiled phrase as de-

scribed in Section 4.1.4. Underscores represent universally-quantified variables in a Theorem whose specific values are irrelevant.

The first assumption of Theorem 3 states that running the CVM on a list of commands compiled from the Copland phrase t produces a trace c of events. The remainder is identical to the conclusion of Theorem 2. Note that the event index i serves as the link between the CVM and LTS event semantics: In run_cvm as the initial IO event ID incremented at each invocation of an attestation-relevant event; in $\mathcal{V}$ to annotate the phrase t, which in turn denotes the Event System reference semantics for events.

5.1.1 Lemmas

To prove Theorem 3, it is enough to prove intermediate Lemma 4 that relates event traces in the CVM semantics to those in the LTS semantics. Lemma 4 states that any trace c produced by the CVM semantics is also exhibited by the LTS semantics. We can combine Lemma 4 transitively with Theorem 2 to prove the main correctness result, Theorem 3. Notice the initial evidence type et is relevant in all of these properties: the structure of evidence determines parameters of events emitted by the CVM and LTS executions alike.

Lemma 4 (CVM_Refines_LTS_Events)

run_cvm *(*copland_compile *t)*

{ st_ev := (_, et), st_pl := p, st_trace := [ ], st_evid := i }

$\Downarrow$

{ st_ev := (_, _), st_pl := p, st_trace := c, st_evid := _ }

$\Rightarrow\ \mathcal{C}(\mathsf{t}_{(\mathsf{i})}, \mathsf{p}, \mathsf{et}) \overset{\mathsf{c}}{\rightsquigarrow}{}^{*}\ \mathcal{D}(\mathsf{p}, \mathcal{E}(\mathsf{t}, \mathsf{p}, \mathsf{et}))$.

Lemma 4 rules out any "extra" CVM event traces not captured by the LTS semantics. It is worth pointing out that we could extend the CVM semantics with additional system events and still prove Theorem 3 directly. This is because Theorem 3 only mentions the *ordering* of *attestation-relevant* system events captured by Event Systems derived from $\mathcal{V}$. However, indirection through the LTS semantics is a convenient refinement because of its closer compatibility with fine-grained CVM execution. The proof of Lemma 4 proceeds by induction on the Copland phrase t that is to be compiled and run through the CVM. Each case corresponds to a constructor of the phrase grammar and begins by conservative simplification and unfolding of run_cvm. Each case ends with applying a semantic rule of the LTS semantics.

A second core property of the CVM is that it transforms Copland Evidence consistently with the LTS semantics, stated as Lemma 5. Recall that $\mathcal{E}$ is a denotation function indicating the reference semantics for Copland evidence. Similar in structure to the proof of Lemma 4, the proof of this Lemma proceeds by induction on the input Copland phrase t. We will see later how this property is critical for an appraiser that must rely on precise cryptographic bundling and the shape of evidence produced by a valid CVM.

Lemma 5 (CVM_Preserves_Evidence_Type)

If run_cvm *(*copland_compile *t)*

{ st_ev := (_, et), st_pl := p, st_trace := [ ] } ⇓

{ st_ev := (_, et′), st_pl := p, st_trace := c } *then*

et′ = $\mathcal{E}$(t, p, et).

Because we cannot perform IO explicitly within Coq, we use st_trace CVM field to accumulate a trace of calls to components external to the CVM. This trace records every IO invocation occurring during execution and their relative ordering. Lemma 6 says that st_trace is irrelevant to the remaining fields that handle evidence explicitly during CVM execution. This verifies that erasing the st_trace field from CVM_st is safe after analysis.

Lemma 6 (st_trace_irrel)

If run_cvm *(*copland_compile *t)*

$\{$ st_ev $:=$ e, st_pl $:=$ p, st_trace $:=$ tr_1 $\}$ $\Downarrow$

$\{$ st_ev $:=$ e′, st_pl $:=$ p′, st_trace $:=$ _ $\}$ *and*

run_cvm *(*copland_compile *t)*

$\{$ st_ev $:=$ e, st_pl $:=$ p, st_trace $:=$ tr_2 $\}$ $\Downarrow$

$\{$ st_ev $:=$ e″, st_pl $:=$ p″, st_trace $:=$ _ $\}$ *then*

e′ $=$ e″ *and* p′ $=$ p″.

Another key property upheld by the CVM is that event traces are *cumulative*. This means that existing event traces in st_trace remain unmodified as CVM execution proceeds. Lemma 7 encodes this vital "distributive property" over trace suffixes that is used in several other Lemmas to simplify trace decomposition.

Lemma 7 (st_trace_cumul)

If run_cvm *(*copland_compile *t)*

{ st_ev := e, st_pl := p, st_trace := m ++ k } ⇓ *st′ and*

run_cvm *(*copland_compile *t)*

{ st_ev := e, st_pl := p, st_trace := k } ⇓ *st″ then*

*(*st_trace *st′)* = m ++ (st_trace *st″*).

5.1.2 Automation

There are many built-in ways to simplify and expand expressions in Coq. Unfortunately, most expand terms too far or not enough. To reach a middle-ground we define custom automation in Ltac, Coq's proof tactic language. First we define a custom "unfolder" that carefully expands primitive monadic operations like bind and return, along with CVM-specific helpers like invoke_ASP.

Next we define a larger simplifier that repeatedly invokes the targeted unfolder followed by cbn and other conservative simplifications. This custom simplification is the first step in most proofs and is repeated throughout as helper Lemmas transform the proof state to expose more reducible expressions. We also leveraged the StructTact [69] library, a collection of general-purpose automation primitives for common Coq structures like match and if statements, originally developed for use in the Verdi [55, 68] framework. Combined with our custom automation this makes the proofs robust against small changes to the CVM implementation and greatly simplifies proof maintenance after more significant refactoring.

5.2 Appraisal Correctness

Given the generalized appraisal procedure from Section 4.2.2, we now define and justify its correctness. Formal treatment is warranted because errors in the unpacking of evidence bundles can lead to misplaced trust in the system being appraised. We decompose appraisal correctness into two main properties: *appraisal coverage* and *appraisal soundness*. Coverage ensures precise segments of the raw binary evidence have the expected cryptographic integrity and checks against golden measurements, while soundness characterizes the strength of an appraisal result with respect to the target system being appraised.

5.2.1 ASP Coverage

ASP Coverage says that primitive checks performed during appraisal combine to exhaustively cover each ASP measurement in the original attestation request. The inference rule in Figure 5.1 defines a relation between appraisal evidence results (values of type E_{Tc}) and Copland Events called *covers_meas*. The intuition behind "e *covers_meas* ev" is: appraised evidence e incorporates a primitive check of the evidence produced by measurement event ev during attestation. There are two ways to satisfy this relation, indicated as the expanded definitions asp_covers_meas and hash_covers_asp.

In these definitions, $\in_T$ and $\in_C$ are sub-term relations for the E_T and E_{Tc} evidence representations, respectively. asp_covers_meas thus ensures that a primitive ASP check occurs as *direct sub-evidence* of the appraisal result e. Notice that the parameter i embedded within the call to do_asp matches the ASP event ID in the conclusion of *covers_meas*. This confirms the connection between the

$$\mathsf{asp\_covers\_meas} := (\mathsf{ASP_c}\ \bar{a}\ p\ (\mathsf{checkASP}\ \bar{a}\ (\mathsf{do\_asp}\ \bar{a}\ p\ i))\ \_)\ \in_C\ \mathsf{e}$$

$$\mathsf{hash\_covers\_asp} := (\mathsf{ASP_E}\ \bar{a}\ p\ \_)\ \in_T\ \mathsf{e_T}\ \wedge\ (\mathsf{HSH_c}\ q\ (\mathsf{checkHash}\ \mathsf{e_T}\ \mathsf{q}\ \mathsf{bs})\ \mathsf{e_T})\ \in_C\ \mathsf{e}$$

$$\frac{\mathsf{asp\_covers\_meas}\ \vee\ \mathsf{hash\_covers\_asp}}{\mathsf{e}\ covers\_meas\ (\mathsf{ASP_{event}}(i, p, \bar{a}, \_))}\ covers\_meas$$

Figure 5.1. *covers_meas* inference rule.

unique measurement event instance during CVM execution and the raw binary result passed to appraisal.

hash_covers_asp also covers an ASP event, but indirectly through a call to checkHash. Recall that in the CVM the hash primitive *consumes* any raw ASP evidence collected up to that point and compacts it into a single fixed-length hash value. Although in an implementation a hash is cryptographically one-way, during verification we track the *type* of evidence being hashed. This allows us to state in hash_covers_asp that a specific ASP measurement with Evidence Type $\mathsf{ASP_E}\ \bar{a}\ p\ \_$ appears as a sub-evidence of the raw hash passed to checkHash.

We can now state the appraisal coverage property for ASPs in Lemma 8. This Lemma states that for each ASP measurement event derived from an arbitrary Copland phrase t, proper checks appear in the appraisal result. Notice that we compute and pass the expected Evidence Type to appraise, along with the raw evidence computed by compiling and running t through the CVM.

One of the well-formedness assumptions is the predicate not_none_none over Copland phrases. This predicate serves to disallow sub-phrases of the form: $\mathsf{t1}\ \overset{(-,-)}{\prec}\ \mathsf{t2}$ and $\mathsf{t1}\ \overset{(-,-)}{\sim}\ \mathsf{t2}$. The effect of such a phrase is to erase all evidence accumulated up to that point by forwarding empty evidence to both of the t1 and t2 branch subterms. If permitted, these "evidence-erasing" subterms would pre-

vent CVM executions from being "evidence-cumulative-modulo-hashing". Without this property, appraisal coverage as stated above does not hold: a measurement event could produce evidence during attestation that is erased before appraisal.

Lemma 8 (appraisal_asp_coverage)

$\forall$ t p et bits bits′,

$<$ well_formedness_assumptions $>$,

If run_cvm *(*copland_compile *t)*

{ st_ev := (evc bits et), st_pl := p, st_trace := _ } $\Downarrow$

{ st_ev := (evc bits′ _), st_pl := _, st_trace := _ }

then

$\forall$ ev $= \mathsf{ASP}_{\mathsf{event}}(\mathsf{i}, \mathsf{p}', \bar{\mathsf{a}}, \_) \in \mathcal{V}(\mathsf{t}, \mathsf{p}, \mathsf{et})$,

appraise (aeval t p et) bits′ *covers_meas* ev

5.2.2 Signature Appraisal Coverage

Similar to appraisal coverage for ASPs, we must ensure that digital signatures computed over evidence during attestation are properly checked during appraisal. The definition of a Lemma for signature coverage is similar in structure to ASP coverage, so we leave the details to the Coq development. However, one additional well-formedness constraint is worth mentioning here: disallowing evidence of the form hhc p bs et, where et involves a signature evidence type. Preventing such input and output CVM evidence ensures attestation will never perform a hash over signed evidence.

To see why this poses a problem for generalized appraisal, consider the Copland phrase: $\mathsf{t1} \rightarrow \mathsf{SIG} \rightarrow \mathsf{HSH}$ with a random nonce with ID n passed as initial evidence from the appraising party. This will produce evidence of the form:

hhc p bs (gg p et′) where et′ is the type of evidence produced by executing t1 and et′ mentions the nonce ID n. Recall that appraising hash evidence involves reconstructing the hash value based on the *type* of evidence hashed. Signed evidence of type gg p et′ is generated at attestation-time using the private key of the signing place p. Unless the appraiser is also p, it should not have access to that private key, and thus cannot re-create the signature. The random nonce embedded as initial evidence causes the hash to differ for each run of the protocol, preventing the target from precomputing the signature and passing it to the appraiser as a golden value.

To ensure the appraiser can always reconstruct golden hashes, we limit attestations to those where hashes are performed over primitive measurement values and other hash composites of the same. While seemingly overly-restrictive, it aligns with the intended use of hashing in Copland. Similar to not_none_none, rather than complicating the Copland language definition we introduce predicates over inputs and outputs of the CVM to prevent problematic combinations of Copland phrases and initial evidence that produce such "hashed-signature" evidence that elude appraisal. Discovering and handling corner cases like these early in the system design lifecycle justifies the extra effort involved in formal verification of attestation and appraisal components.

5.3 Verification LOC Statistics

5.3.1 Copland Reference Semantics

	Reference Semantics		
	File Name	LOC	Totals
Implementation	Term_Defs	80	**80**
Specification + Proof	More_lists	300	
	Defs	9	
	Term_Defs	376	
	Term	321	
	Event_system	541	
	Term_system	200	
	Trace	740	
	LTS	800	
	Main	285	**3572**
Automation	Preamble	16	
	AutoPrim	27	
	Defs	67	
	Term_Defs	48	
	Term	61	
	Term_system	26	
	Trace	55	
	Main	12	**312**
			Total LOC: **3964**

5.3.2 CVM (Attestation)

	CVM (Attestation)		
	File Name	LOC	Totals
Implementation	ConcreteEvidence	52	
	Maps	20	
	GenStMonad	54	
	StVM	6	
	MonadVM	170	
	Impl_vm	32	**334**
Specification + Proof	ConcreteEvidence	1230	
	AxiomsIO	33	
	EqClass	45	
	Maps	40	
	Helpers_VmSemantics	346	
	External_Facts	80	
	VmSemantics	1200	**2974**
Automation	ConcreteEvidence	21	
	MonadVM	59	
	Auto	94	
	Helpers_VmSemantics	57	
	VmSemantics	25	**256**
			Total LOC: **3564**

5.3.3 Appraisal

	Appraisal		
	File Name	LOC	Totals
Implementation	StAM	15	
	OptMonad	16	
	Appraisal_Defs	107	
	Impl_appraisal	19	
	Impl_appraisal_alt	33	**190**
Specification + Proof	Appraisal_Defs	439	
	Appraisal_AltImpls_Eq	123	
	Helpers_Appraisal	2816	
	Appraisal	990	**4368**
Automation	AutoApp	99	
	Appraisal_Defs	104	
	Helpers_Appraisal	589	**792**
			Total LOC: **5350**

5.3.4 LOC Totals

By artifact:

- Reference Semantics: 3964
- CVM (Attestation): 3564
- Appraisal: 5350

By code category:

- Implementation: 604
- Specification + Proof: 10914
- Automation: 1360

Total Overall LOC: 12878

Chapter 6

Appraisal Soundness

While the verified Copland-based components provide a foundation for trusting attestation results, components external to Copland also contribute to that trust. Even with strong properties like appraisal coverage it remains unclear what one can safely infer about a target system based on a given appraisal result. The strength of such inferences is at the core of a property that we will henceforth refer to as *appraisal soundness*. Our motivation for discussing appraisal soundness is not to arrive at a formal theorem that declares once-and-for-all that attestation implies absolute security. Even the task of *stating* formal properties like these is complicated by experience in security that shows adversaries will find ways to corrupt any useful system [40, 59, 60], and that layered attestation schemes have inherent trade-offs involving trust [12]. Instead we aim for a more nuanced notion that incorporates a broader view of attestation contexts.

To position appraisal soundness within a broader workflow that involves the design, execution, and analysis of Copland-based attestation scenarios, we introduce the *Copland Verification Architecture* in Figure 6.1. Within this workflow the Appraiser platform takes a Copland phrase t as input and relies on a stateful

execution environment (introduced in Section 4.2.4) to generate and remember a random nonce, construct an attestation session around that nonce, and perform appraisal over the evidence result. The Target platform leverages the CVM to carry out the request and emits an event trace guaranteed to respect the Event System partial ordering derived statically from t. In addition to these core attestation components, appraisal soundness depends on ambient factors of the execution environment including its Security Architecture and the strength of protocols with respect to an active adversary attempting to subvert attestation goals.

6.1 Security Architecture

Alongside the CVM on the Target platform, measurement components m_1, m_2, ..., and target applications a_x, a_y, ..., operate within a *Security Architecture* depicted in Figure 6.1 by dotted green lines around components. The analysis framework remains parameterizable over such *means of isolation* that protect trusted measurers from their potentially-untrusted targets. This supports dropping in alternative isolation mechanisms as deemed appropriate by the consumer of a particular attestation. Access control and inter-component dependencies also contribute to the Security Architecture for a given attestation scenario. In the workflow of Figure 6.1 we can encode such properties in first-order syntax accepted by a model-finder that incorporates them into a higher-level adversarial analysis. Assuming the security mechanisms are properly configured, a final analysis pass may confirm mitigation of certain classes of attacks on the attestation system. In Section 6.4 we instantiate a concrete measurement architecture that leverages the formally verified seL4 microkernel [32] for component separation and analyze its design in the context of a Copland-based attestation.

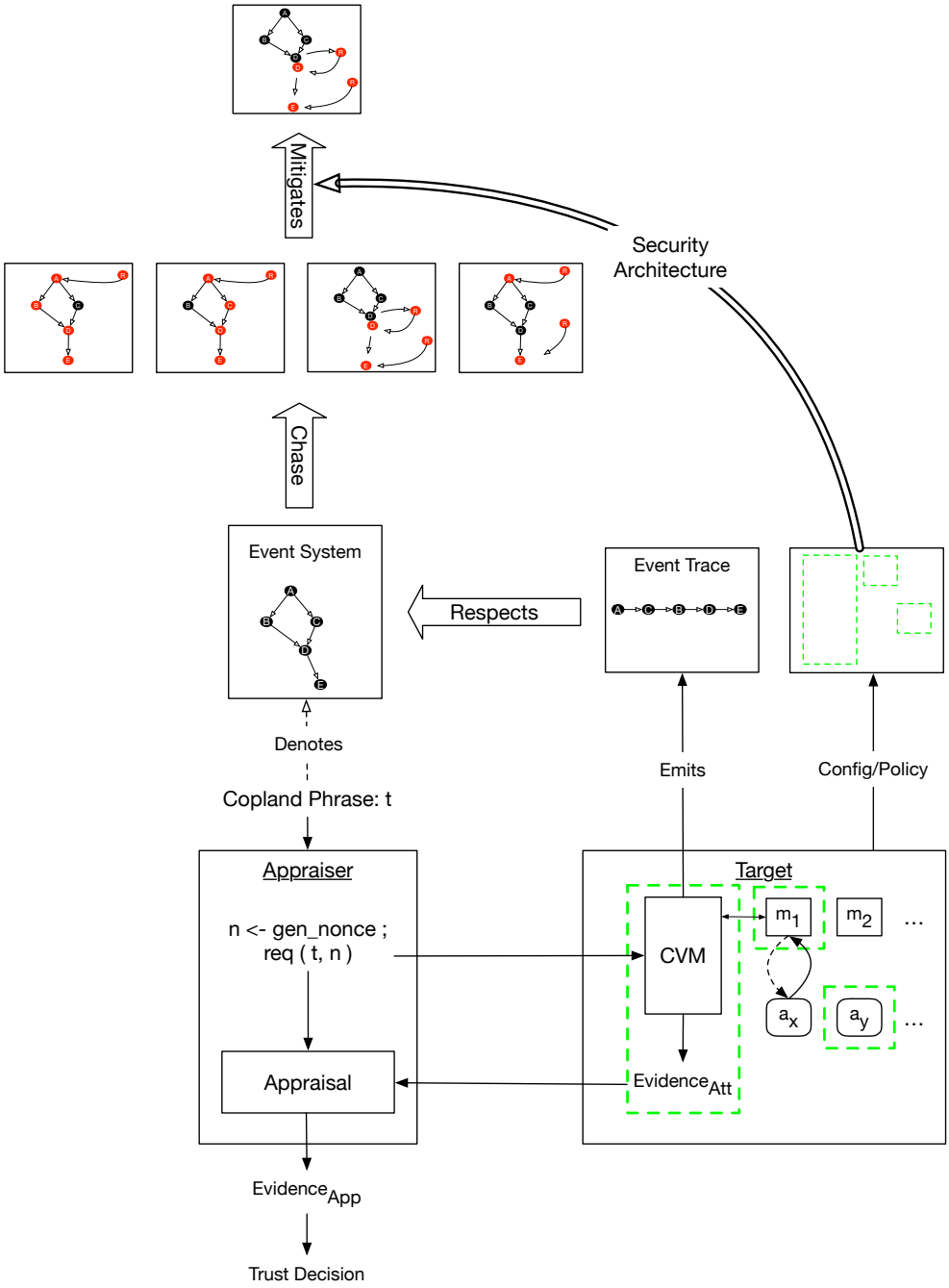

Figure 6.1. Copland Verification Architecture.

6.2 Adversary Analysis

Rowe [60] lays the foundation for formal analysis of active adversaries in a layered attestation context. This formalism models an adversary capable of corrupting and repairing arbitrary measurement components, and gives formal justification for bottom-up measurement strategies that confine an adversary to "recent or deep" corruptions that are known to be more difficult in layered architectures. In more recent work, Rowe et. al. [58] compose the formalisms of a capable adversary with an idealized semantics of honest Copland participants to automate analysis of Copland protocols via novel theory extensions to the general-purpose Chase [57] model-finder.

Leveraging this body of work, we incorporate proofs about CVM execution and appraisal from the current work to discharge axioms made within the analysis workflow in Figure 6.1. The adversarial analysis leverages a model-finding tool developed by our collaborators at MITRE instrumented explicitly to analyze Copland-based protocols [58]. Given a Copland phrase t and an indication of the measurement target(s) of interest, the tool produces an initial set of models that describe all distinct ways an active adversary could corrupt the target and go undetected by measurement. This initial analysis relies on existing axiomatizations of Event Systems and an adversary that are encoded as first-order logical statements understood by the underlying Chase [57] model-finder. The encoding of Event Systems characterizes honest measurements derived directly from t, while the adversarial model encodes rules of a capable attacker that can arbitrarily corrupt and repair the very same measurement components.

One of the goals of a well-designed attestation system is that it should place a *high as possible* burden on the attacker [59, 60]. Towards this goal, our analysis

framework incorporates additional constraints to reflect properties of the security architecture, eliminating certain classes of attack from consideration. More concretely, we encode architectural properties as logical assumptions to the model finder to indicate corruptibility based on the integrity of individual components and their contextual dependencies. With measurement protocol and architectural assumptions incorporated, the attack models that remain must be acceptable to the consumer of attestation. Otherwise, an iteration to refactor the Copland protocol and accompanying security architecture may be in order. Section 6.4 gives representative examples of attack models uncovered in the design of a real-world UAV attestation scenario.

6.3 Component Implementations

While the generality of Copland makes it amenable to higher-level analysis, it also leaves lower-level component implementations under-specified. The core artifacts introduced in this work refine the execution semantics of Copland to a more fine-grained notion of bundling and appraisal of evidence. While the Coq verification environment supports rigorous proofs about their subtle interaction, these artifacts remain "executable" only within this formal setting. In order to extend appraisal soundness to running systems, the task remains to instantiate the components that orchestrate Copland protocols, along with their external dependencies, within a richer executable context.

Towards this goal, we implemented the Haskell Attestation Manager, a collection of libraries written in the functional language Haskell [42]. Haskell comes with a wealth of existing libraries that aid in instantiating a concrete execution environment for Copland that includes communication mechanisms, crypto-

graphic primitives, concurrency support, binary encoding/decoding of inductive datatypes, file I/O, and more. In addition to supporting these "drop-in" services, the latest version of the Haskell AM links to the formally verified core components automatically via Coq's built-in code extraction mechanism. While this garners increased confidence in the resulting attestations, the primary utility of the Haskell AM has been as a rapid prototyping environment to tease apart the design space of integrating formal and non-formal components of an attestation system. Ongoing work [31] builds on these insights in developing an AM in CakeML [36], a language that enjoys a compiler with formal semantics.

With the executable environment of the Haskell AM in hand, we can now come full circle to instantiate the virus checker attestation scenario from Section 2. As it turns out, this is a special case of a more general class of attestation called *Certificate Style* as proposed by Helble et. al. [27]. A Copland phrase for this pattern is repeated here as follows:

```
*P0,n: @P1[(attest P1 sys) ->
         @P2[(appraise P2 sys) ->
               (certificate P2 sys) ]]
```

Here we see a pipeline of ASPs carried out by three participants: place P_0 the Relying Party, place P_1 the Attester and P_2 the Appraiser.

Mapping participant IDs to the virus checker is somewhat straightforward: P_0 becomes the top-level client, P_1 the target platform hosting the virus checker, and P_2 the trusted-third-party appraiser. The main insight for configuring the rest of the phrase is that `attest` can leverage its own "nested" CVM instance to run any Copland-specified protocol that measures the target of interest (here the virus checker and its operational environment). If we likewise instantiate

appraise with our generalized appraisal procedure, the Appraiser place is prepared to evaluate the potentially intricate evidence structure bundled by attest. Finally, the certificate ASP could be a simple digital signature over the appraised evidence, or a more complicated summary of the same. Such a certificate strategy benefits from the formal semantics of Copland since the appraisal result is an E_{Tc} value, a convenient structure for post-processing appraisal results.

The codebase of the Haskell AM prototype, along with the full details of the Flexible Mechanisms pattern instantiations, are freely available on GitHub [54]. An early version of this prototype is outlined in Petz and Alexander [51], however the current version has undergone significant refactoring in response to evolution of the Copland language itself, its formalization, and other more traditional software engineering considerations like enhanced modularity and usability. We also present more details of the Haskell implementation and instantiate the remaining Flexible Mechanisms patterns from Helble et. al. in Chapter 7.

6.4 Case Study: DARPA UAV Demonstration Platform

In order to exercise the Copland Verification Architecture workflow in a real-world case study, we will now give an overview of the demonstration platform of an Unpiloted Air Vehicle (UAV) system taken from our work on the DARPA CASE program. Our task was to extend a preexisting legacy implementation with support for layered attestation built with Copland Attestation Managers, and formally analyze the resulting system. This new architecture will exhibit desirable security properties that we can leverage as sound assumptions in the analysis framework in Figure 6.1.

The scenario features two communicating systems, a ground station and a

UAV. The UAV accepts flight plans from the ground station in the form of waypoints and attempts to navigate within security constraints. Both the ground station and UAV run preexisting UxAS [3] software, running in Linux. In the unhardened system the UAV has no assurance that the ground station it accepts directions from is trustworthy. The UAV has no means of distinguishing a compromised or fake ground station from a genuine, trustworthy one. To address these concerns, we extend the two systems to support Copland attestation protocols.

6.4.1 Ground Station and UAV Security Architectures

The hardened security architectures for the UAV and groundstation platforms appear in Figures 6.3 and 6.2, respectively. The UAV platform listens for commands from a groundstation, then consults with the onboard attestation manager component (AM) to determine whether or not the commands are coming from a trustworthy ground station. If appraisal succeeds, the Filter component caches the identity of the trusted ground station and accepts its subsequent commands for some user-defined time window. The Ground Station architecture involves two attestation managers at different layers of the system called the User AM and Platform AM. The User AM sits inside a Linux VM environment and is responsible for application-specific measurements of UxAS,and also accepts attestation requests from the UAV platform. The Platform AM runs as a native seL4 component and is responsible for measuring the neighboring linux environment. The formal guarantees of the seL4 microkernel ensure that the Platform AM is completely isolated from its potentially-corrupted target of measurement.

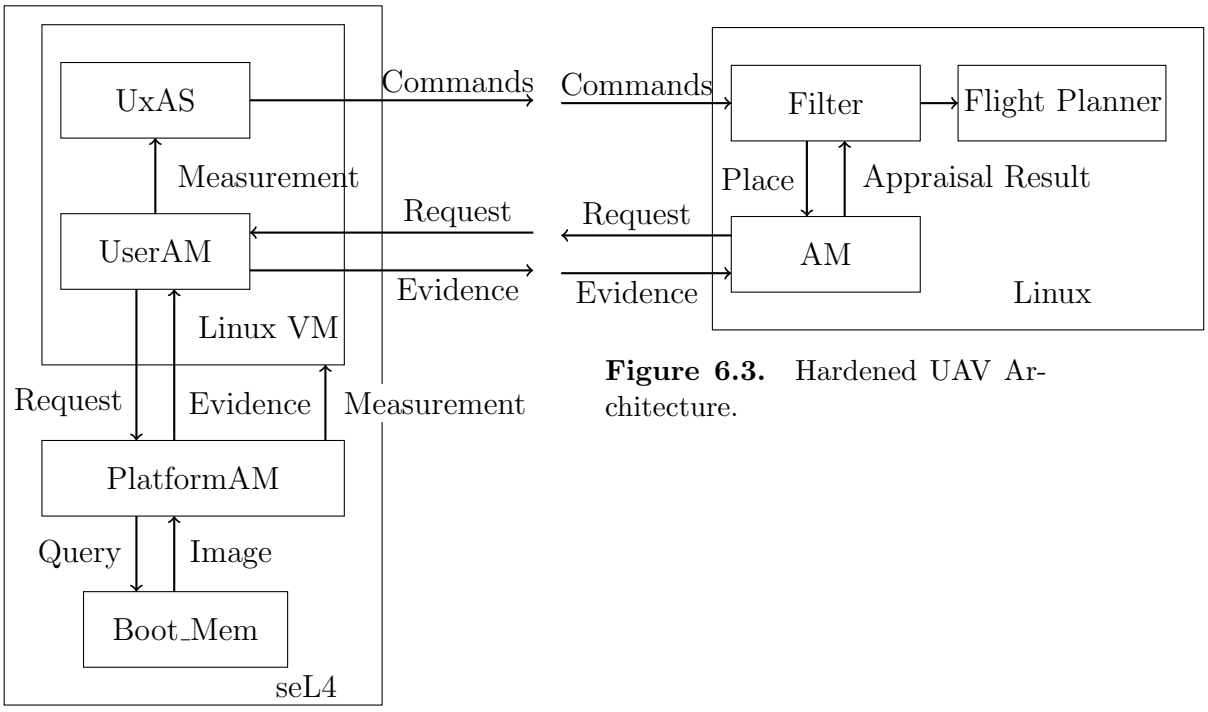

Figure 6.2. Hardened Ground Station Architecture.

Figure 6.3. Hardened UAV Architecture.

6.4.2 Copland Phrase Description/Components

A Copland phrase to measure the transformed ground station target platform appears in Figure 6.4. The phrase begins with: $*\mathsf{heliAM}, \mathsf{n}$: which designates heliAM (the UAV AM component from Figure 6.3) as the *appraising place* and also specifies a nonce n be passed as initial evidence with the attestation request. Next, @userAM[...] specifies that the terms inside [...] be executed at the userAM place. userAM and platAM are place identifiers that represent distinct attestation domains both running on the remote ground station (UserAM and PlatformAM from Figure 6.2).

As soon as userAM receives the initial request, @platAM[...] specifies that it initiate a request to the (more privileged) Platform AM. platAM starts by invoking query_img to read the contents of the seL4 image img loaded at boot-time into

```
*heliAM, n:
    @userAM [
              @platAM [ (query_img bootMem img) ->
                          ((kim userAM ker)
                                  +~+
                           (uim userAM uam)) -> !
                         ] ->
              ((uam userAM uxas_ctxt)
                      +~+
               (uam userAM uxas)) -> !
              ]
```

Figure 6.4. UAV Copland Phrase

protected memory. This image serves as evidence of the static configuration of all components on the ground station platform at startup. The abstract nature of places in Copland allows us to represent this protected memory region as its own place identifier bootMem that will be incorporated into analysis.

After querying the image, platAM performs cross-domain measurements of components at userAM. One is (kim userAM ker) that specifies an integrity measurement of the linux OS kernel running at userAM. The other is (uim userAM uam) that specifies an integrity measurement of the uam (Userspace Attestation Manager) component that itself performs more specialized measurements of UxAS. The +~+ operator specifies that both of its subterms may execute in parallel, whereas -> requires strict linear sequencing.

After receiving evidence of platform integrity from platAM, userAM proceeds to perform specialized measurements of the target application. These measurements perform dynamic monitoring of the execution context of the UxAS flight planning software and UxAS itself. After completing its measurement, userAM signs the accumulated evidence bundle and sends it in a response back to heliAM.

6.4.3 Event Semantics

Figure 6.5 shows the Event System, a partial ordering on measurement events, determined by the Copland phrase in Figure 6.4. This graphical output comes from the model finder tool, but an identical partial ordering can be derived from the Copland formal semantics in Coq. This ordering states that the boot image is queried first, followed by the integrity measurements launched from platAM. The integrity measurements are free to execute in any order, but must complete before the specialized userspace measurements at userAM begin. This event ordering is the first input into the automated attack analysis in the model finder, and its soundness is justified by the Copland Virtual Machine that is formally verified to uphold such measurement orderings.

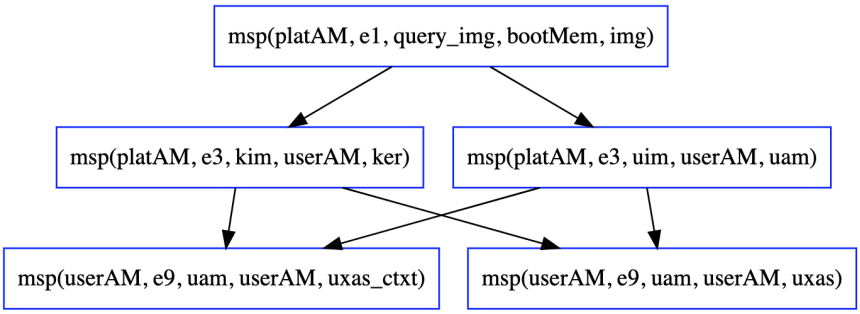

Figure 6.5. Event System derived from the Copland phrase in Figure 6.4 above.

6.4.4 Architectural Assumptions

Given the measurement events in Figure 6.5, the model finding tool requires additional assumptions about the environment in which these measurements are carried out before it can produce a meaningful analysis. The first of these is an

indication of the *measurement event(s) of interest*. In other words, the instant(s) during attestation where we would like to determine if the adversary has sufficiently corrupted a specific target while avoiding detection. Figure 6.6 shows this statement for our UAV attestation scenario, encoded in first-order syntax accepted by the model finder. This logical statement asks for models where either uxas or something in its execution context uxas_ctxt are corrupt (phi predicate) after the event where uam measures uxas.

```
l(E) = msp(userAM, e, uam, userAM, uxas)
  => prec(E,E2) &
     phi(userAM, uxas, E2) | phi(userAM, uxas_ctxt, E2).
```

Figure 6.6. Assume adversary avoids detection at main measurement event.

Given only honest measurement events and this event of interest, the model finder will generate an exhaustive set of attack models assuming a capable adversary that can corrupt and repair arbitrary components on the system. One such model appears in Figure 6.7. Here we see the adversary has corrupted the userspace AM (uam) after it was measured, and leveraged it to lie about the corrupted state of uxas_ctxt. It later covers its tracks by repairing uam. In total, the tool generates 74 *essentially distinct* attack models like this. Because many of these attacks are unacceptable, analysis like this early in the design of an attestation protocol is useful to pinpoint parts of the system that may require hardening.

Further assumptions come from properties about dependencies of components and the architecture where they operate. The statements in Figure 6.8 encode that components at bootMem and platAM do not depend on any other components

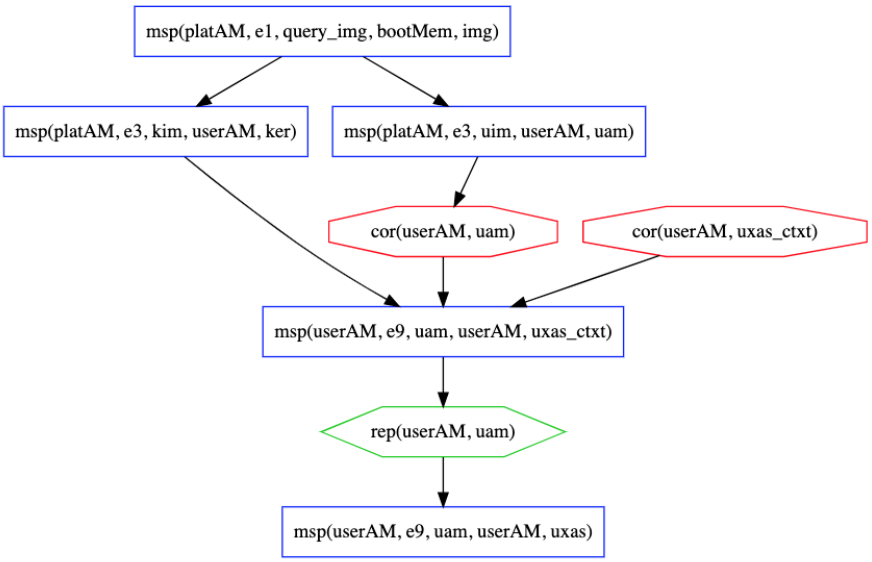

Figure 6.7. One attack model: corruption and repair of uam

for their own integrity. Without assumptions like these the analysis conservatively assumes that for each component, there exists an arbitrary component co-resident at its place capable of affecting its integrity. Each of these statements are justified by trust in specific components and their environment: bootMem is a protected storage location; measurement components at platAM run in an isolated, native seL4 environment with limited dependencies and limited-purpose code.

```
% No dependencies for components at bootMem or platAM
ctxt(bootMem, C, C2) => false.
ctxt(platAM, C, C2) => false.
```

Figure 6.8. Contextual assumptions about "deep" components in the architecture.

Statements in Figure 6.9 encode additional assumptions about the corruptibility of components. The first says that only way to corrupt a component at platAM is by corrupting img at bootMem. The final two statements say that the

boot image cannot become corrupted, and that the only way for uam to become corrupted is via a corrupted OS kernel. The first of these is justified by protecting the image somewhere like trusted hardware or a dedicated seL4 component, where only highly-privileged boot loader code has write access. To justify the latter, in our prototype we limit the code of uam to very specific measurement functions and include them as a library packaged with the Copland Virtual Machine at userAM.

```
% platAM components only corrupted via a corrupt boot image
l(E) = cor(platAM, C) => phi(bootMem, img, E).

%% img in bootMem cannot be corrupted
phi(bootMem, img, E) => false.

% user AM (uam) only corrupted via a corrupt kernel
l(E) = cor(userAM, uam) => phi(userAM, ker, E).
```

Figure 6.9. Assumptions about the "corruptibility" of components.

Figure 6.10 lists four final assumptions that eliminate most of the remaining feasible attack models. The first three make explicit the context of the remaining components in userspace. The final assumption ignores attacks on the kernel. Because dynamic attacks on OS kernels are feasible in practice, one could remove this assumption to explore the implications of such an attack. However, for our final analysis we assume that the (kim userAM ker) measurement launched from platAM gives sufficient evidence that the kernel will run uncorrupted long enough for the other measurements that depend on it to complete.

Given all of the above assumptions, the attack model in Figure 6.11 characterizes one of the two remaining ways to corrupt uxas and go undetected (an analogous model exists for corrupting uxas_ctxt). Specifically, the adversary must

```
% All components at userAM (except ker itself)
% depend on ker
ctxt(userAM, C, uam) => C = ker.
ctxt(userAM, C, uxas_ctxt) => C = ker.

% In addition to ker, uxas depends also on uxas_ctxt
ctxt(userAM, C, uxas) => C = ker | C = uxas_ctxt.

% Ignore attacks that corrupt ker
l(E) = cor(userAM, ker) => false.
```

Figure 6.10. Final Architecture Assumptions

corrupt uxas (or its context) *after* they are measured and before the flight planning services are consumed. This is an example of a recent or time-sensitive attack that is in theory more difficult to execute [60]. In what follows, we discuss strategies for making these types of attacks more difficult still by repeated measurement, thus justifying them as acceptable attack models.

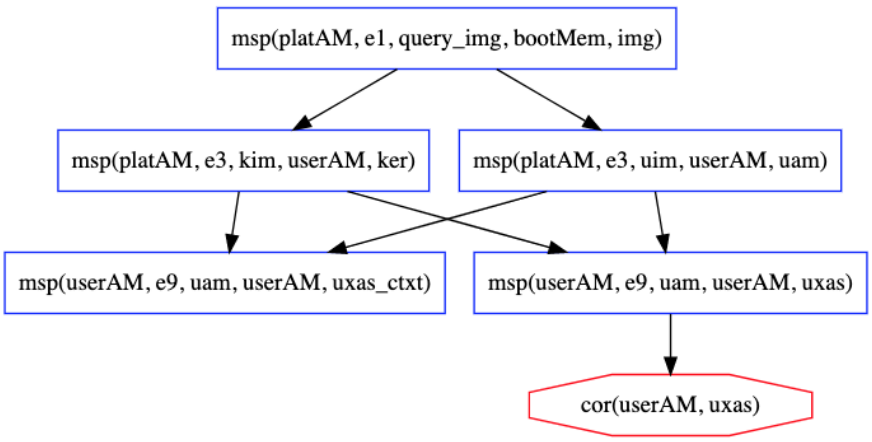

Figure 6.11. Final Attack Model

6.4.5 AM Monad alternatives

From the appraiser's perspective a Copland protocol is executed atomically before evidence bubbles back to its environment. While it is possible to craft standalone phrases that are sufficient for simple, static attestation scenarios, there are cases where an appraiser might desire more flexibility to orchestrate the execution of multiple Copland phrases and compose intermediate evidence results. Pure Copland has no persistent state and no error handling mechanism to account for failed or divergent attestations. To address these issues we deploy the Attestation Manager (AM) Monad environment of Section 4.2.4.

An example computation in the AM Monad called `attest_gs` prepares, executes, and appraises a Copland phrase from the UAV scenario as follows:

```
attest_gs t :=
  do {n  <- generateNonce;
      ev <- run_cvm(n,t);
      b  <- appraise n t ev;
      update_filter(b)}
```

For flexibility we parameterize `attest_gs` by the Copland phrase `t` intended to measure the groundstation target. Assuming the phrase from earlier in Figure 6.4 is assigned the name `case_cop`, we could instantiate this AM Monad computation via function application: `attest_gs case_cop`.

Earlier we saw that even with strong architectural protections and bottom-up measurement strategies, an adversary can escape detection by performing timely attacks on components after they are measured. One way to make these attacks less effective is to perform periodic re-measurement of the system:

```
do_while(true) (
  attest_gs case_cop;
  sleep(s))
```

Here `s` is within some time interval chosen by the appraiser to make attack and repair of the target difficult for the adversary. Depending on the scenario and capabilities of the attacker, `s` could be bounded under a certain threshold, or even randomly generated within an acceptable range.

However, a problem arises if the time required to complete all measurements of the phrase exceeds `s`. Recall that the `case_cop` phrase involves measurements of deep components that may require significant time and system resources. In real-time embedded systems with hard scheduling requirements, attestation services might be resource-constrained to complete their measurement tasks [11]. This limitation is at odds with deeper measurements that tend to stall or freeze the system to capture its state. When full appraisals are too costly, performing one deep measurement of the system during initialization followed by repeated, shallow probes may be sufficient to establish a baseline and maintain evidence of integrity. Partitioning `case_cop` into its deep and shallow portions has the form:

```
case_deep :=
    @userAM [ @platAM [ (query_img bootMem img) ->
                        ((kim userAM ker)
                                +~+
                         (uim userAM uam)) -> ! ]]

case_shallow :=
    @userAM [ ((uam userAM uxas_ctxt)
                      +~+
                (uam userAM uxas)) -> ! ]
```

A final AM Monad computation that performs a deep attestation of the ground station at frequency `s`, and shallow attestations at frequency `r` is as follows:

```
do_while(true) (
  attest_gs case_deep;
  do_for_duration(s) (
    attest_gs case_shallow;
    sleep(r)
  )
)
```

The AM Monad environment is a standard state and exception monad with a formal definition in Coq [50] and prototype implementations in CakeML and Haskell [31, 54].

Chapter 7

Instantiating Flexible Mechanisms

So far we have introduced a collection of formal artifacts core to attestation and appraisal of Copland-based protocols. The Copland Virtual Machine of Section 4.1 takes an attestation term specified in Copland along with initial evidence, and produces an evidence bundle that is input-compatible with the generalized appraisal procedure of Section 4.2. The Copland Verification Architecture of Chapter 6 lifts the formal properties of these artifacts into a higher-level analysis framework based on model finding to characterize an active adversary's ability to thwart a given attestation goal. Finally, in Section 6.4 we instantiated this pipeline with a specific attestation scenario tailored to an existing demonstration platform under the DARPA CASE. effort. While this served as an illustrative proof-of-concept for how to leverage the pipeline to gain confidence in a singular attestation *design*, it left unresolved how an implementation might incorporate the formal components as *executable artifacts*, together with non-formal components, to support a diverse collection of attestation patterns.

In this chapter we begin to fill in these gaps by introducing the Haskell Attestation Manager prototype implementation and demonstrate how it supports a larger collection of attestation shapes, namely the *Flexible Mechanisms* attestation scenarios proposed in the work of Helble et. al. [27]. These shapes were classified by a group of experts in attestation, and arose in part to comply with and influence guidance of an IETF working group focused on Remote ATtestation procedureS (RATS) [5]. Given their origins, these attestation patterns serve as an evolving benchmark that any attestation framework should aim to support.

7.1 Haskell Attestation Manager

The Haskell Attestation Manager is a collection of libraries written in the functional language Haskell [42] that provide a concrete execution environment for Copland attestation protocols. Because Copland was designed to run on heterogeneous systems, its protocol descriptions leave implementation details abstract as parameters. These configurable items include communication mechanisms, measurement procedures, cryptographic primitives, and concurrency support. Haskell comes with a wealth of existing libraries to meet these needs including TCP/IP, JSON, cryptography, multi-threading, shared memory, binary encoding/decoding, file I/O, and more. An early version of this prototype is outlined in Petz and Alexander [51], however the current version has undergone significant refactoring in response to evolution of the Copland language itself, its formalization, and other more traditional software engineering considerations like enhanced modularity and usability. The Haskell AM codebase is freely available on GitHub [54].

In addition to supporting the instantiation of Copland's "drop-in" services, the current version of the Haskell AM links these to the formally verified core

components automatically via code synthesis. Coq's code extraction mechanism supports automatic translation of the core attestation components defined in Gallina (Coq's built-in functional language) to executable code in Haskell. While this garners increased confidence in the resulting attestations, the translation itself lacks formal guarantees about how semantics are preserved from Gallina to Haskell. Setting aside the advantages of such "foundational" guarantees, the rest of this section outlines the useful outcomes of the design and development of the Haskell AM that emerge even without total correctness. In particular, we focus on its primary utility as a rapid prototyping environment to tease apart the design space of integrating formal and non-formal components of an attestation system.

The act of prototyping such a system forces implementation decisions that concretize details left abstract in its formalization. Furthermore, these decisions inform two complimentary avenues of design. The first is a software architecture that influences subsequent (perhaps more formal) attestation manager implementations. There is ongoing work that brings patterns from the Haskell AM to one written in CakeML [36], a language that (unlike Haskell) enjoys a formal semantics down to metal. The other avenue is identifying parts of the system that may warrant further formalization. Because a prototype must concretize details left abstract in its formalization, additional complexity may be uncovered, leading to more opportunities for bugs. This under-specification is apparent in mechanisms that orchestrate execution of parallel Copland phrases outlined in Section 7.1.5.

7.1.1 Admitted Definitions in Formal Spec

The first step towards integrating the formal components in Coq with the executable Haskell prototype is to fill in definitions left abstract in the former. The

standard way to omit definitions in Coq is the Admitted keyword. Interpreted in the context of a *proof*, Admitted acts as an axiom, or an assumption of some fact without explicit proof. From a computational and code synthesis perspective Admitted serves as an empty definition, assumed to be instantiated after verification in a separate executable environment. From this perspective such a definition is *uninterpreted* during verification; its concrete implementation is delayed until code extraction. These hollow definitions in the specification represent items that are either too low-level to warrant formalization, or stand-in for calls to external functions like I/O procedures that are unnatural to define in Coq explicitly.

Here we will consider Admitted definitions in two distinct categories: datatypes and functions. An example of an Admitted datatype is the type for raw binary data evidence called BS (short for "Byte String"):

```
Definition BS : Set.  Admitted.
```

This Coq syntax means that we are defining a *type* called BS, to be used in a computational setting (in the universe Set), with no concrete definition (Admitted). With this uninterpreted type in hand, we can use it to construct larger datatypes that depend on it:

```
Definition RawEv := list BS.
```

Here the type for raw evidence is defined as a sequence of BS values. Because the list datatype is defined first class in Coq we can reason directly about how the CVM bundles evidence into RawEv structures during attestation, all while referring to the BS type by name alone (see more details in Section 4.1).

As one example of an uninterpreted function, consider encodeEvRaw:

```
Definition encodeEvRaw(e:RawEv): BS.  Admitted.
```

The type signature indicates a transformation from a sequence of BS values to a single BS value. This is invoked within the CVM (and likewise during appraisal) to compact its internal evidence structure in preparation for cryptographic operations like signing or hashing that expect a single binary value as input. The choice to leave encodeEvRaw uninterpreted in Coq is due to its low-level nature, and because its implementation depends on concrete instantiation of the BS datatype.

To instantiate BS in Haskell, we chose the ByteString type from the popular Data.ByteString library [14]:

```
module BS where
import qualified Data.ByteString as B (ByteString, ...)
type BS = B.ByteString
```

A ByteString is a time and space efficient representation of binary data, encoded internally as Word8 arrays in C. ByteStrings are standard in Haskell libraries that handle binary data, including most communication and cryptographic APIs. A convenient instantiation for encodeEvRaw is the concat function from Data.ByteString that simply concatenates the raw evidence sequence into one long ByteString value. Although verification of attestation and appraisal never require unfolding this definition, we can still ensure within the Coq environment that encodeEvRaw is called at the correct time and over appropriate raw evidence parameter values.

While the above Admitted definitions stand in for low-level manipulation of binary data, other kinds of Admitted definitions stand in for IO operations that

invoke services external to the core attestation process. These *IO stubs* invoke ASP measurement routines, cryptographic libraries, communication sessions with remote CVMs, and parallel CVM instances. Some of the parameters to these stubs are themselves uninterpreted datatypes in Coq:

```
Definition Plc, ASP_ID, TARG_ID, Arg: Set.  Admitted.
```

These represent configuration parameters passed via Copland phrases, for example as parameters to ASP procedures:

```
Inductive ASP_PARAMS: Set :=
| asp_paramsC: ASP_ID -> (list Arg) -> Plc -> TARG_ID -> ASP_PARAMS.
```

The only assumption made about these datatypes in the Coq specification is that they have decidable equality:

```
Definition eq_aspid_dec:  forall x y: ASP_ID, {x = y} + {x <> y}.
Proof.  Admitted.
```

We arbitrarily instantiate the first three of these (Plc, ASP_ID, TARG_ID) with the Haskell type Int, and Arg with the Haskell type String. Full Coq type signatures for the uninterpreted IO stubs appear in Figure 7.1.

Each ASP and cryptographic function (do_asp, do_sig, do_hash) ends with a parameter of type Event_ID. This parameter is used purely for verification to log each invocation as a unique attestation event. As such, we can erase that parameter upon extraction using Coq's Extraction Implicit command:

```
Extraction Implicit do_asp [3 4].
```

```
Definition do_asp (params :ASP_PARAMS) (e:RawEv) (mpl:Plc) (x:Event_ID) : CVM BS.
Definition do_sig (bs:BS) (p:Plc) (sigTag:Event_ID) : CVM BS.
Definition do_hash (bs:BS) (p:Plc) (hshTag:Event_ID) : CVM BS.
Definition do_start_par_thread (loc:Loc) (t:Term) (e:RawEv) : CVM unit.
Definition do_wait_par_thread (loc:Loc) (t:Term) (p:Plc) (e:EvC) : CVM EvC.
Definition doRemote_session (t:Term) (pTo:Plc) (e:EvC) : CVM EvC.
```

Figure 7.1. Coq type signatures for Admitted IO stubs.

The above will erase the third and fourth parameters of do_asp at each of its call sites in the extracted code. Here the place tag parameter is also irrelevant during concrete execution. Specifying each IO stub return type as a CVM monad computation allows their invocation inline within the Copland Compiler implementation. Recall that the CVM monad in Coq uses a stateful record to manage evidence and track events during verification. We will see later in Section 7.1.4 how to leverage Haskell's monad transformer library to *extend* the CVM monad in the extracted implementation to acquire a read-only environment, error, and IO functionality, all without modifying the stateful evidence bundling.

7.1.2 Deriving typeclass instances in Haskell

Notice in the above instantiations how the uninterpreted datatypes and functions are organized into distinct modules. This module structure is derived directly from the original Coq definitions, and plays nicely with Coq's code extraction. Coq's SeparateExtraction directive preserves module boundaries upon extraction, thus allowing a strategy where extracted modules (with empty definitions) can be ignored and replaced wholesale with their concrete Haskell counterparts. These concrete module instantiations are then linked with other extracted and non-extracted code, where module imports are handled automatically for extracted code based on the import hierarchy in the original Coq definitions.

While the core implementations of attestation and appraisal components can be extracted directly from Coq and integrated without modification into the Haskell AM, other components require a small amount of wrapping or extending to play nicely with Haskell's typeclass system. In particular, we often wish to pretty-print or read representations of Copland datatypes, which requires them to conform to Haskell's Show and Read typeclass instances. For our prototype it is also often necessary to encode (decode) datatypes to (from) their binary and JSON representations. We can leverage Haskell's Typeclass Deriving mechanisms to automatically derive such instances for most datatypes. This supports a strategy where we extract datatype definitions directly from Coq, then add a "wrapper module" that simply decorates these definitions with Deriving clauses. Deriving JSON instances requires additional custom configuration, but we leave further discussion of JSON component interfaces to Section 7.2.

7.1.3 ASP Servers

Because Copland was designed with generality in mind, measurement and cryptographic services remain abstract in protocol specifications as simple identifiers. However, during protocol execution these identifiers must be mapped to concrete implementations. In the Haskell AM we instantiate such services external to the CVM with a common abstraction called an *ASP Server*. An ASP Server has a unique address and listens for requests sent to that address. Upon receiving a request, the server runs a custom handler depending on its role. Each handler has the following general type:

```
(FromJSON a, ToJSON b) => (a -> IO b)
```

Its purpose is to accept a request message of type a, perform some IO action,

and return a response message of type b. Also notice that the input type must have a FromJSON typeclass instance, and likewise the output type a ToJSON instance. Such a JSON interface is crucial for generality, and to support re-use and interoperability of ASPs implemented in diverse language environments.

In the Haskell AM, these servers are implemented as Unix Domain Servers, where each socket address derives its namespace from the file system. In the Copland language, the notion of "address" is lifted to more abstract Place identifiers. However in a concrete implementation each CVM instance must be configured with mappings from Place IDs to concrete addresses before execution. In addition to ASP servers there are other external services required by a CVM that include cryptographic implementations, communication sessions to remote CVMs, and local interaction with parallel CVM servers. Configuration of these services is discussed briefly in Section 7.1.4 and JSON interfaces to these components are outlined in Section 7.2.

7.1.4 Instantiating the CVM Monad

In the formal specification the Copland Virtual Machine is implemented within an environment called the the CVM Monad. In Coq this environment is defined from scratch as a monolithic state and failure monad, and is instrumented with proof automation tactics to unfold monadic statements in a principled way. Although the state and failure effects are sufficient to confirm important properties of the CVM during verification (event trace ordering, evidence bundling, etc.), a richer computational context is required to carry out concrete executions of Copland protocols. In particular, the protocols require a collection of static (read-only) configuration parameters and also require access to an external IO environment

to carry out measurement and cryptographic tasks.

Contrary to Coq's somewhat rigid verification environment, Haskell comes with a standard and well-supported mechanism to glue together a collection of diverse computational effects: monad transformers [38]. Monad transformers allow us to build a monadic type incrementally via a "stack" of monadic transformers, where each transformer adds a class of effect. Since we already defined the CVM as a state monad in Coq, we will need to map that functionality to the State Monad Transformer (StateT) in Haskell. In addition, we will need to add an immutable configuration context (ReaderT) and an IO environment (IO Monad). The top-level Haskell type for this transformer is as follows:

```
import qualified Control.Monad.State.Lazy as SL
type St s = SL.StateT s COP
```

The type for St says it is a computation that supports stateful operations over a structure of type s, and is composed with an underlying monad COP (COP will be introduced shortly). Recall that in Coq we implemented St as a state monad from scratch with the standard primitives like ret(return), bind, put, and get. We then instantiated the CVM monad in Coq with the CVM state structure cvm_st:

```
Definition CVM := St cvm_st.
```

To leverage the transformer functionality in Haskell we alias each monadic primitive of the St monad in Coq with its Haskell counterpart. The names of these primitives, defined in Figure 7.2, must match those defined in formal environment. This allows synthesized code to link properly with the surrounding Haskell prototype. Although our goal for the prototype is not foundational correctness, we can be confident that the monadic primitives for St in Coq behave

the same as their counterparts in Haskell. With these primitives shadowed in Haskell, the synthesized CVM code will be able to reference them purely by name and benefit from the robust built-in monadic support.

```
ret :: a -> CVM a          bind :: CVM a -> (a -> CVM b) -> CVM b
ret = SL.return            bind m f = m SL.>>= f

put :: s -> CVM ()         get :: CVM s
put = SL.put               get = SL.get
```

Figure 7.2. Shadowing of monadic primitives in Haskell.

While the COP monad is not referenced at all in the formal specification, we can leverage the compositionality of monad transformers in Haskell to add underlying Reader and IO functionality to concrete CVM executions:

```
import qualified Control.Monad.Reader as RT

type COP = RT.ReaderT Cop_Env IO
```

The base-level IO monad provides access to the external input/output environment: access to files, networks, system entropy, etc. The ReaderT transformer supports read-only access to a static, immutable environment, configured before the monadic computation begins. Purely within Haskell we instantiate this environment as a record structure called Cop_Env in Figure 7.3 that contains configuration items needed during interpretation of Copland phrases in the CVM.

While some of these fields are simple flags that aid in prototyping and debugging (simulation, debug), others provide crucial pointers to components external to the CVM during Copland phrase execution. One example is the nameServer field which maps Place identifiers to concrete addresses, providing guidance for

```
data Cop_Env =
  Cop_Env { simulation :: Bool,
            debug :: Bool,
            nameServer :: M.Map Plc Address,
            sig_mechanism :: Sign_Mechanism,
            asp_sockets :: M.Map ASP_ID Address,
            parServer :: Address }
```

Figure 7.3. The Cop_Env environment for read-only configuration of the CVM.

@ term interpretation. Fields like sig_mechanism and asp_sockets map abstract cryptographic and measurement primitives to concrete services during execution. It is the responsibility of each platform owner to populate these mappings to support a given Copland phrase. While such a registration process is largely out of scope for the current work, we describe some related experiments in Section 7.1.6. Finally, the parServer field points to a local server that handles parallel execution of Copland phrases, which is discussed in more detail in the following subsection.

7.1.5 Parallel Interpretation of Copland Phrases

In the formal specification of the CVM, interpretation of parallel Copland phrases (terms involving $\overset{\pi}{\sim}$) is modeled as an asynchronous interface to an assumed external CVM instance responsible for running phrases in a parallel thread of control. The core of this interface is specified in Coq as follows:

```
Definition do_start_par_thread (loc:Loc) (t:Term) (e:RawEv) : IO unit.

Definition do_wait_par_thread (loc:Loc) : IO RawEv.
```

do_start_par_thread specifies a Copland phrase and initial raw evidence sequence to run in parallel, and also a Loc identifier that indicates a location to listen for the evidence result. do_wait_par_thread models the corresponding block-waiting

operation to extract evidence from a given Loc. Because Copland is designed to run in diverse environments, this interface leaves the type of Loc abstract, and also does not make any assumptions about specific models of concurrency supported by the underlying execution environment.

To see how these two functions interact, recall the general shape for compilation of parallel Copland phrases within the CVM:

```
copland_compile (abpar_par loc sp t1 t2) :=
  (e1,e2) <- split_ev sp
  do_start_par_thread loc t2 e2 ;;
  put_ev e1 ;;
  copland_compile t1
  ...
  e1' <- ...
  e2' <- do_wait_par_thread loc
  join_par e1' e2'
```

Notice first that the parallel term being compiled is preannotated with a memory location loc used to share evidence between the main and parallel CVM threads. Critically, the same loc value is passed to both the start and wait commands to facilitate execution of the t2 subterm while the t1 subterm is evaluated locally. These loc annotations are populated by an input list provided in a preprocessing phrase before compilation. In the prototype we acquire this list before compilation from a parallel CVM server, and rely on assumptions that each memory location in the list is unique and initially empty. In the current Coq specification there are no formal constraints on the list besides that it has sufficient size to completely

annotate the phrase. Proofs about the formal semantics only rely on matching loc parameters between the start and wait commands.

The parallel CVM server is responsible for maintaining a pool of available locations that act as shared memory cells to hold evidence passed between the main and parallel CVM threads. Although the main thread could spin its own child threads to avoid a separate parallel server altogether, decomposing the system in this way is more general and stays compatible with architectures that do not have first-class multithreading. This style of message-passing concurrency is more general and common in embedded/real-time systems architectures.

```
reserve_locs :: Term -> CVM [Locs]
reserve_locs t := do
  tSize = thread_count t
  (AckInitMessagePar locs) <- par_server_session (InitMessagePar tSize)
  return locs

handle_par_init :: IO ()
handle_par_init := do
  (InitMessagePar tSize) <- receive
  atomically $ do
    ls = read locs_var
    if (length ls < tSize)
      then retry
      else do
        (locs, rest) = partition ls tSize
        locs_var := rest
        send (AckInitMessagePar locs)
```

Figure 7.4. Pseudocode for reserve_locs and handle_par_init.

Assuming a parallel CVM server has access to a pool of shared evidence cells, it must be prepared to handle two types of requests from the main CVM thread. The first is an initialization request that arrives before annotation and compila-

tion of a Copland phrase. The goal of the initialization request is to reserve a list of Loc values for phrase annotation. Pseudocode for the client and server implementations appear above in Figure 7.4. reserve_locs first calculates the number of parallel CVM threads required to execute a given Copland phrase, then initiates a session with the parallel CVM server to reserve a collection of Locs of appropriate size. handle_par_init represents the server-side logic, where it receives the term size as a message, checks for availability of Loc cells, and sends them in response.

The two remaining types of requests a parallel CVM server must handle correspond to the start and wait commands launched from the main CVM client thread. Their pseudocode appears in Figure 7.5.

```
handle_par_start :: IO ()
handle_par_start := do
  (StartMessagePar loc t e) <- receive
  e' <- run_cvm (t, e)
  store[loc] := e'

handle_par_wait :: IO ()
handle_par_wait := do
  (WaitMessagePar loc) <- receive
  v := store[loc]
  case v of
    Just e ->
      ls = read locs_var
      locs_var := ls ++ [loc]
      delete loc store
    Nothing -> retry
```

Figure 7.5. Pseudocode for handle_par_start and handle_par_wait.

handle_par_start accepts a request to run a Copland phrase in parallel and store the result evidence at a specific shared memory location. handle_par_wait retrieves

an evidence value from shared memory and performs appropriate maintenance of the shared memory store. While these pseudocode snippets provide insight into the semantics of parallel CVM execution, their implementations fall outside of the formally verified attestation components described in this work. In particular, there are minimal guarantees about the semantics of shared memory when multiple CVM instances must interact with the same parallel server. We leave the formal modeling and verification of such properties to future work.

7.1.6 Configuration of CVM Nodes and ASPs

A prerequisite for executing Copland-based protocols is configuration of each protocol participant with every service that it will need during execution. While the primary role of the formal CVM semantics is to *trace and analyze* invocations of these services, the current specification assumes the availability of each service prior to execution. In a concrete implementation these services not only require instantiation, but other components must be configured with a means to contact them and request their services. The types of services required by the CVM during execution include ASP, cryptographic, and external CVM instances (both local parallel and remote). While we introduced these types of services individually in Section 7.1.3, the different strategies for instantiating them has yet to be discussed.

In the current prototype there are two main methods of instantiation for services. The first method is *local spawning*. In this method the top-level protocol participant is responsible for spawning services as local background threads of control. This process involves walking the Copland phrase(s) involved in a scenario and deriving a set of nodes and their configuration such that each phrase involved has the services it needs to complete execution. While the distributed

nature of attestation scenarios in the real world makes this type of configuration less realistic, its primary advantage is for rapid prototyping of end-to-end examples. The other spawning method is *distributed manual spawning* that requires each node and service be configured and spun up individually, perhaps even by separate platform owners if the nodes are in remote attestation domains. While this is a more realistic execution context for attestation protocols, it takes much more effort and coordination to start up and tear down services during testing.

Although these methods are at opposite ends of the "automatic configurability" spectrum, the Haskell AM prototype also offers an important middle-ground in the design space. First is a feature called the "default ASP" that designates an ASP for handling all uninstantiated ASPs with a dummy implementation. If an ASP is not mapped to a concrete address in the configuration, it will be mapped to this default address. This approach supports an incremental strategy where ASP implementations can be added one-by-one while testing the larger attestation evidence flow. Another configuration option in this same spirit is to specify that only the CVM nodes themselves be locally spawned, but require ASP implementations not mapped to the default ASP be started manually. This alternative focuses on incremental testing of ASP implementations and evidence pipelines while avoiding complete configuration of CVMs. This approach also has the advantage of isolating debugging output to individual nodes rather than interleaved in the top-level thread.

The Haskell AM highlights important points in the design space for configuring CVM services that is simply absent from the formal specification. However it is still biased toward ease of prototyping and as a testing ground for new features of Copland. In particular, challenges remain to provide convenient and

reliable ways to configure these systems in real-world distributed attestation environments. Nonetheless, ongoing work that involves attestation managers in different programming language environments like CakeML [31], and a more diverse attestation testbed environment are already borrowing design decisions and configuration patterns from this work.

7.2 Copland + JSON

Because components in our attestation manager prototype must communicate structured data over general-purpose communication channels, we have designed a JSON interface for interaction with different Copland-based services. The types of messages exchanged include ASP and cryptographic servers, CVM attestation sessions, and parallel CVM requests. In what follows we give representative examples of the JSON interfaces. Complete specifications are included in Appendix A.

7.2.1 General ADT JSON Schema

We represent Copland-based language terms as Algebraic Data Types (ADTs) in the Haskell prototype. In JSON we represent ADTs as objects with two fields:

1. constructor-the constructor name as a JSON string ($<$ string $>$). Constructor names must be unique for unambiguous parsing.

2. data-An ***ordered*** JSON array ($<$ array $>$) that holds the arguments for that particular constructor. Members of the data array will differ from constructor to constructor.

The general schema for ADTs (labelled by placeholder $<$ ADT $>$) is as follows:

```
{
  "constructor": < string >,
  "data": < array > | < ADT >
}
```

where the < ADT > alternative in the data field accounts for degenerate nesting of constructors (for example in the ASP constructor below).

7.2.2 Copland JSON Schemas

The JSON object schemas for Copland phrases (t in Figure 3.1), Evidence Types (E_T in Figure 3.5), and Typed Concrete Evidence (E_{Tc} in Figure 4.7) appear in their entirety in Appendix A. Representative examples of schemas for Copland phrase constructors that satisfy the < term > placeholder are as follows:

< asp_params > := [< number >, [< string >], < number >, < number >]

```
{
  "constructor": "Coq_asp",
  "data": {"constructor": "ASPC",
           "data": < asp_params >
           }
}
```

```
{
  "constructor": "Coq_asp",
  "data": {"constructor": "SIG"}
}
```

```
{
  "constructor": "Coq_att",
  "data": [ < number >,
            < term > ]
}
```

< SP > := “ALL” | “NONE”

```
{
  "constructor": "Coq_bseq",
  "data": [ [< SP >, < SP >],
              < term >,
              < term > ]
}
```

where < number > and < string > are placeholders are for the standard JSON number and string datatypes. The order of items in the “data” subarray is significant–they match the order of arguments to each constructor.

$$
\begin{aligned}
RequestMessage = \{ & \\
toPlace &:: \mathsf{p}, \\
fromPlace &:: \mathsf{p}, \\
reqNameMap &:: \mathsf{p} \Rightarrow \mathsf{Address}, \\
reqTerm &:: \mathsf{t}, \\
reqEv &:: [\ \mathsf{bs}\ ] \ \}
\end{aligned}
$$

$$
\begin{aligned}
ResponseMessage = \{ & \\
respToPlace &:: \mathsf{p}, \\
respFromPlace &:: \mathsf{p}, \\
respEv &:: [\ \mathsf{bs}\ ] \ \}
\end{aligned}
$$

Figure 7.6. Request and Response Message record structures.

7.2.3 Remote CVM Message Schemas

Request and Response Messages are record structures in Figure 7.6 that facilitate communication with CVMs. Their respective JSON object schemas are:

```
{
  "toPlace": < number >,
  "fromPlace": < number >,
  "reqNameMap": < nameMap >,
  "reqTerm": < term >,
  "reqEv": < raw_evidence >
}
```

```
{
  "respToPlace": < number >,
  "respFromPlace": < number >,
  "respEv": < raw_evidence >
}
```

where fields that hold raw data (bs parameters in the grammar) are base64-encoded JSON strings that hold binary values–hashes, nonces, signatures, etc:

< b64_string > := < string > (Base64 encoded)

< raw_evidence > := [< b64_string >]

< nameMap > is a JSON object of the following form:

$\{pl_1{:}addr_1, pl_2{:}addr_2, ...\}$

where pl_1, pl_2, ... are JSON key strings that represent a Copland place identifier (i.e. "1", "2", ...) and $addr_1$, $addr_2$, ... are JSON strings (< string >) that represent platform addresses. We leave address strings abstract in this specification, but a common usage would be a string of the form ip:port.

7.3 Flexible Mechanisms Implementation

Now that we have introduced the software architecture of the Haskell AM and how it integrates the core attestation components extracted from Coq, we can move on to instantiate the *Flexible Mechanisms* patterns of attestation proposed by Helble et. al. [27]. Each pattern is specified as one or more Copland phrases, whose definitions and corresponding architecture diagrams are repeated below for the sake of self-containment.

The primary remaining task to instantiate these patterns is to provide implementations for primitive ASPs that are only described abstractly and labeled by name in [27]. The core attestation components and surrounding Haskell AM toolset are then sufficient to instantiate each of the attestation patterns in the Flexible Mechanisms benchmark.

7.3.1 Certificate Style (Simple)

The first pattern is called Certificate Style, and its general shape can be seen in the diagram of Figure 7.7. Here an entity called the Relying Party (place P_0) sends an initial request containing a nonce n to the Attester (place P_1). Upon receipt of the request, the Attester collects some evidence and bundles the result with the nonce to produce evidence(n), which it forwards to the Appraiser (place P_2). After evaluating the evidence, the Appraiser returns a certificate also bound to the nonce n to the Attester, who concludes the protocol by relaying the certificate(n) back to the Relying Party.

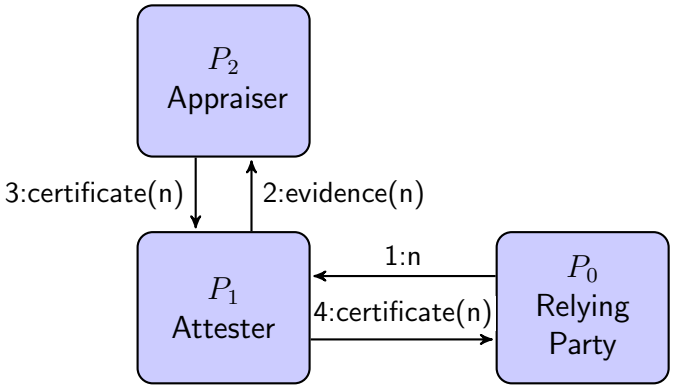

Figure 7.7. Certificate-Style. Fig. 5 on pg. 29:15 of [27].

Although a certificate might be a more sophisticated operation, in its simplest form it is a simple digital signature over the evidence. In that case, a Copland phrase like the following is sufficient:

```
*P0,n: @P1[(attest P1 sys) ->
         @P2[(appraise P2 sys) -> ! ]]
```

Here the digital signature primitive "!" serves as the certificate, and if we assume place P_2 is properly configured to handle CVM attestation requests, then we

know it is equipped with such a signature capability. If all places involved are configured properly, we can also trust them to invoke the specified services and bundle evidence properly. What is left then is to instantiate the attest and appraise ASP procedures at places P_1 and P_2, respectively. If we initially map both attest and appraise to the default ASP server, the phrase runs as expected in the Haskell AM, producing a top-level evidence bundle at place P_0. While simple, this phrase exercises CVM node configuration by spawning CVM instances at place P_1 and P_2, interpreting @ with concrete socket-to-socket communication mechanisms, and bundling of cryptographic evidence.

7.3.2 Certificate Style

A slightly more sophisticated variation on the Certificate Style pattern involves replacing the digital signature with a custom certificate ASP procedure. The Copland phrase for this is as follows:

```
*P0,n: @P1[(attest P1 sys) ->
         @P2[(appraise P2 sys) ->
               (certificate P2 sys) ]]
```

While this requires a small amount of extra configuration to stand-up the certificate ASP at place P_0, the overall shape is unchanged. If we again instantiate certificate with a dummy implementation, this phrase immediately runs successfully as before. However, our pipeline of ASPs has become interesting enough that the semantics of primitive evidence values is now significant. Because the semantics of the linear sequence operator $\rightarrow$ in Copland is "evidence-cumulative", the evidence produced by attest at place P_1 feeds into appraise at P_2, and likewise from appraise into certificate.

While more sophisticated implementations of ASPs support custom attestation goals, they might also add complexity by breaking the abstraction of otherwise opaque evidence blobs passed between them. To tame some of this complexity we can instantiate the attest ASP such that it leverages the existing Copland and CVM infrastructure. The main insight is to give attest its own CVM instance capable of running any Copland-specified protocol to measure the target of interest (presumably at place P1). This approach allows attest to be configured, perhaps even dynamically, by a subset of the protocol participants. Because the CVM is designed to execute arbitrary Copland phrases, this adds significant expressivity to the Certificate Style pattern.

An important design decision for attest is how to acquire the Copland phrase that runs within it. One option is for a subset of the protocol participants to negotiate the phrase *a-priori* before the Relying Party sends the initial request. In this case, the phrase can be added as a static parameter to attest before runtime. Depending on privacy constraints of the involved participants the phrase could be added either by the Relying Party as a part of the initial request (static arguments to the attest and appraise ASPs), or configured locally by the Attester and known only to the Relying Party by name. A second option is to allow the first action within attest to be an independent negotiation between the Attester and Appraiser domains on-demand. Both of the above strategies are supported by the Haskell AM. Because negotiation is largely out of scope for the current work, we leave the action of choosing a particular phrase within attest abstract as the action choose_phrase which potentially interacts with the IO environment. A pseudocode rendering of the attest implementation appears in Figure 7.8.

```
attest :: [BS] -> IO BS
attest e :=
  t <- choose_phrase
  e' <- run_cvm (t, e)
  return (encode (AttestResult t e') )
```

Figure 7.8. Pseudocode for the attest ASP.

The type of attest reflects the main task of an ASP server implementation: take a raw evidence sequence as input and produce a single binary value as output, potentially interacting with external components via the IO monad. The return value is encoded as a binary blob to be bundled by the CVM in the standard way, and thus available for decoding in subsequent ASPs in the pipeline.

The AttestResult return type is a custom Haskell record datatype that encodes the necessary metatdata to serve as glue between attestation and appraisal:

```
data AttestResult = AttestResult
  { term_ran :: Term,
    ev_res :: RawEv } deriving (Show,Read,Generic)
```

The first field holds the Copland phrase dynamically selected and executed by the Attester. The second is the new raw evidence sequence returned by the CVM after executing that phrase.

Next we must instantiate the appraise ASP such that it can interpret the evidence output by attest. Recall from Section 4.2 that the generalized appraisal procedure has precisely that capability, but requires an expected evidence shape in addition to the raw evidence. Using the evidence denotation function from Section 3.3.1 we can derive the shape of evidence given the shape of the initial evidence passed to attest and also the Copland phrase it selected to run. For simplicity in this scenario, the Appraiser assumes the initial evidence provided

to attest is always a nonce value with ID 0. The phrase t and the raw evidence it produced can be extracted from the AttestResult structure generated by attest. The pseudocode for appraise appears in Figure 7.9.

```
appraise :: [BS] -> IO BS
appraise e :=
  [hd, ..., n] <- peel_bs e
  (AttestResult t attRes) <- decode hd
  evidence_type = eval (t, P1, N 0)
  nonceMap = [(0,n)]
  appRes :: EvidenceC = gen_appraise (evidence_type, attRes) nonceMap
  return (encode appRes)
```

Figure 7.9. Pseudocode for the appraise ASP.

Upon receiving this raw evidence from attest at P_1, appraise at P_2 must first peel off and decode the AttestResult blob. It can then configure and run the generalized appraisal procedure, resulting in a Typed Concrete Evidence result that summarizes the appraisal.

Notice here that we must configure the generalized appraisal procedure with a mapping that includes the nonce value passed initially as input to attestation. This is required to ensure to the Relying Party that the attestation is sufficiently fresh. We also assume the Appraiser knows the precise location of the nonce value within the raw evidence (at the last position in the list). The appraise ASP concludes by encoding the appraised evidence structure as a raw binary blob. Figure 7.10 shows how evidence flows through the attest, appraise, and certificate ASPs. Notice how the overall evidence structure extends according to the CVM bundling semantics, but the semantic content of each ASP call encoded into individual evidence cells.

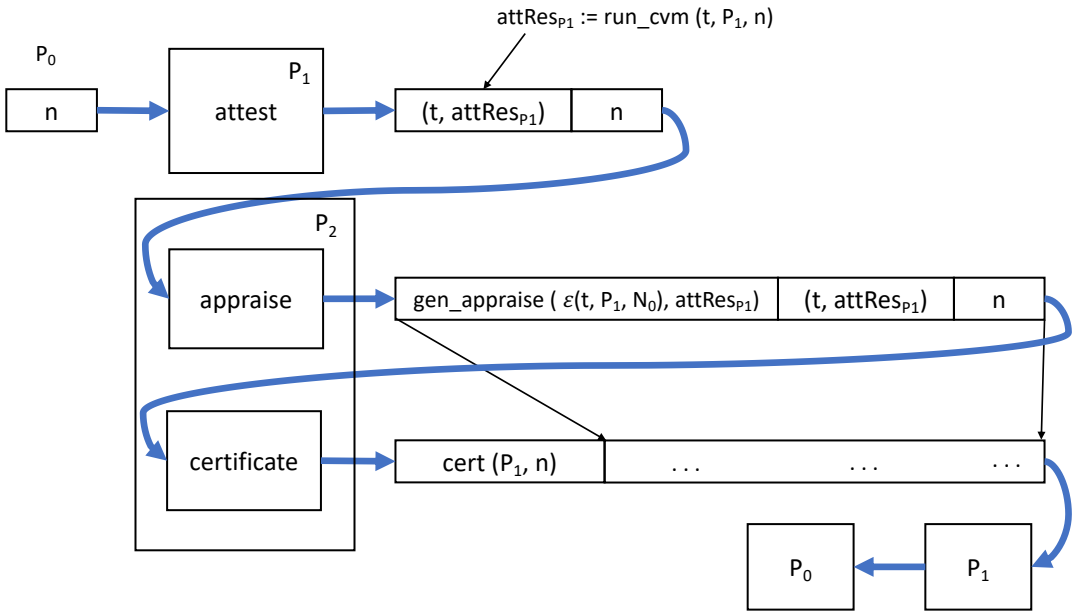

Figure 7.10. Diagram of the attest → appraise → certificate pipeline, ASP evidence encoded as blobs within Copland raw evidence sequence.

The final ASP in the pipeline, certificate, must examine the appraisal result and provide some sort of endorsement of its contents for review by the relying party. As discussed previously, this endorsement might be a simple digital signature over the evidence blob. However, if the certifier is aware of the Copland-based semantics of the prior appraisal result (encoded as Type Concrete Evidence), it becomes feasible to walk that structure and make a more sophisticated judgment by considering sub-appraisals. Said another way, a convenient way to implement our certificate authority is to support a go/no-go decision based on individual appraisal results embedded within the ConcreteEvidence structure. Even with this general strategy in place, many interesting decisions remain about how to walk the structure, which pieces contribute to a certification decision, and how

to report that decision back to the Relying Party. Nonetheless, the predictable semantics of Copland terms and evidence, along with the off-the-shelf attestation and appraisal procedures, provide a solid foundation and rapid testing-ground for such custom certificates.

```
certificate :: [BS] -> IO BS
certificate e :=
  [hd, ..., n] <- peel_bs e
  (ec :: EvidenceC) <- decode hd
  b = certWalk_EvidenceC ec
  bs = bool_to_bs b
  return (sign (bs, n, P1))
```

Figure 7.11. Pseudocode for the certificate ASP.

The pseudocode for the certificate ASP implementation appears in Figure 7.11. This ASP starts similarly to appraise by peeling off the relevant data items from the raw evidence sequence. This time, the front evidence cell is the EvidenceC structure produced by appraisal. With this value in hand and decoded, we invoke the helper function certWalk_EvidenceC which implements a depth-first walk over the EvidenceC structure and returns a boolean result. There are many ways to implement such a walk, and in fact this provides a convenient interface to configure certificates with different levels of scrutiny. For concreteness in this prototype, we implement a simple walker that checks that all appraisal results can be interpreted as "passing". For this particular appraisal, this is effectively an accumulation of boolean checks. After computing this certification result, we can encode it to binary, then sign a certificate package and return it. While there are more options for what is included in this package, here we choose to include the encoded boolean result, the initial nonce value, and the place whose attestation we are endorsing

(note that the place ID could be derived from the EvidenceC structure, or hard-coded if this ASP is configured statically).

7.3.3 ASP Bundling Semantics

Implementing the Certificate Style pattern raised an issue related to the bundling semantics of ASPs performed within the CVM. In the Copland reference semantics evidence is treated abstractly, and is closer in spirit to an evidence *type*. In particular, it is left underspecified how an ASP routine consumes its input evidence, and how it bundles its result evidence with the existing raw evidence sequence. While the evidence type semantics tags each ASP to record that it *potentially* incorporates its input evidence, it leaves the details of this to the semantics of the individual ASP, and correspondingly its interpretation to the complimentary appraisal procedure for that ASP. While this provides greater flexibility for specifying custom attestation scenarios, it creates a potential for ambiguity in the raw evidence sequence derived from the CVM bundling semantics, and correspondingly the generalized appraisal procedure.

For simplicity, the CVM semantics thus far appends all ASP results to the front of the raw evidence sequence. While this is fine for some ASPs that merely gather evidence, it doesn't always make sense for other ASPs that might summarize or cryptographically transform their input evidence. This is the case for the appraise and certificate ASPs of the Certificate Style pattern. There is also the issue of privacy: it may be the case that the Attester platform wishes to keep the details of its measurements and their evidence results confidential and inaccessible by the Relying Party. Notice in Figure 7.10 that under the existing CVM semantics, the final evidence value produced by the certificate ASP includes evidence cells for

results of both the attest and appraise ASPs. Further, by the semantics of the @ phrase, this evidence implicitly bubbles back through place P1 and P0 (Attester, Relying Party) as each place finishes its work. To protect its confidentiality, the Attester platform may wish to negotiate that its results only be visible to the trusted Appraiser.

This scenario uncovers a need for each ASP to be configurable by how it consumes and outputs evidence. In some cases it should be *destructive*: consume the input evidence and replace it completely in the output. In other situations it should append the newly-gathered evidence to existing evidence. Both of these bundling strategies are desirable at different moments in the certificate style pattern: the attest ASP must leave its input nonce untouched, while the certificate ASP should consume and replace its entire input to keep both the attestation protocol and its evidence result confidential.

To address this shortcoming of the existing CVM bundling semantics, we propose a minor extension to Copland that allows distinguishing between destructive and non-destructive ASP terms. Making this distinction first class in the language is vital to the generalized appraisal procedure that depends on the precise structure of raw evidence. This change should only add a minor additional proof burden: primitives already exist that destroy and preserve input evidence (HSH and SIG, respectively). However we leave the details of this update to the language and CVM proofs for future work.

7.3.4 Cached Certificate Style

The Cached Certificate Style is a slight variation on the Certificate Style that adds two new ASPs called store and retrieve that act on a shared evidence cache

at place P1. A diagram of this pattern appears in Figure 7.12, and it can be described in Copland as follows:

```
*P1:(attest P1 sys) ->
    @P2[(appraise P2 sys) -> (certificate P2 sys)] ->
    (store P1 cache)

*P0,n:@P1[((retrieve P1 cache) -<+ _) -> !]
```

Here we see the pattern differs from normal Certificate Style in that it has two top-level places that orchestrate the execution of two separate Copland phrases. The first phrase run by P1 starts with the same attest, appraise, certificate pipeline from before. But instead of simply letting the chain of evidence bubble back to the Relying Party, here P1 uses the store ASP to save the certificate evidence in a local cache. The second phrase is a request from the Relying Party, along with a nonce for freshness, for P1 to retrieve the same certificate from its cache and endorse it with a signature.

With the attest, appraise, certificate pipeline of ASPs already implemented from the previous pattern, the Haskell AM already has nearly all it needs to run this new pattern. The two things that remain are instantiations for the store and cache ASPs, and also a means to run (and configure) two top-level phrases simultaneously. To implement the store and retrieve ASPs, we consolidate them into a single ASP called cache and allow it to accept the store and retrieve commands as distinct arguments. This strategy simplifies configuration because we only need to instantiate a single mutable memory cell within the cache ASP server that holds the cached raw evidence sequence. Because the store and retrieve commands may access the same cache server simultaneously, we implement the

cache memory cell as a Haskell TVar from its Software Transactional Memory library [65]. This ensures atomic access to its contents, and also supports retrying access when for example the retrieve command occurs before the cache has been populated.

With the cache ASP configured, what remains is configuring and running the two phrases simultaneously. However the assumption there was that only one top-level Copland phrase needed support. Running two phrases is accomplished by instantiating two separate AM Monad instances, and configuring services that result from taking the union of parameters from both of the Copland phrases involved. The relevant parameters for configuring services are the places involved (to configure CVM instances) and the ASPs involved. Once the union of these places and ASPs are configured and running, running the above phrases as separate threads (where retrieve waits for store to populate the cache) results in a successful run of this pattern.

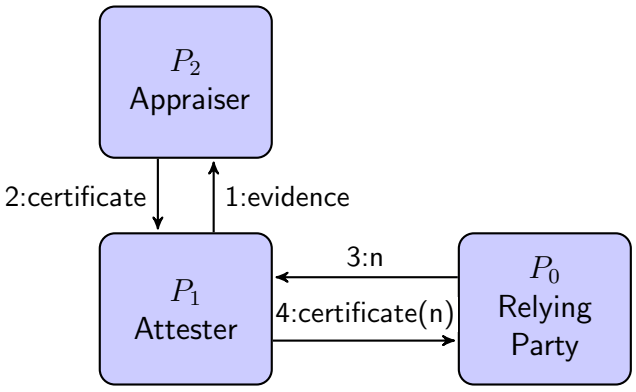

Figure 7.12. Cached Certificate-Style. Fig 6 on pg. 29:16 of [27].

7.3.5 Parallel Mutual Attestation

The Parallel Mutual Attestation pattern is depicted in Figure 7.14, with corresponding Copland phrase in Figure 7.13.

```
*P0,n0: @P1[(attest01 P1 sys)] ->
        @P2[(appraise01 P2 sys)]

*P1,n1: @P0[(attest10 P0 sys)] ->
        @P2[(appraise10 P2 sys)]
```

Figure 7.13. Copland phrase for Parallel Mutual Attestation.

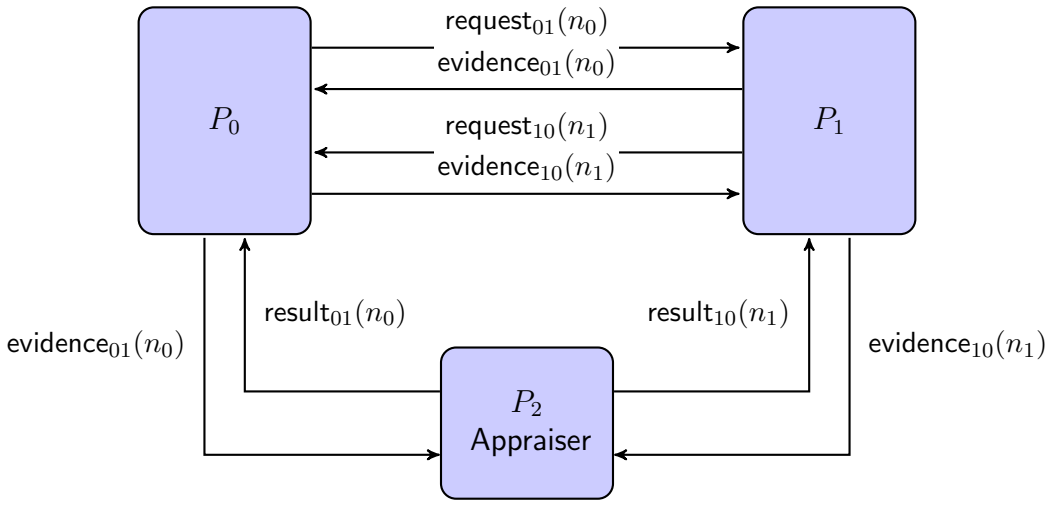

Figure 7.14. Parallel mutual attestation. Fig 8 on pg. 29:18 of [27].

The aim of participants P_0 and P_1 is to acquire evidence about the trustworthiness of one another, then have a trusted third party Appraiser at place P_2 appraise that evidence. The Flexible Mechanisms implemented so far provide a sufficient toolbox to configure and run this scenario almost immediately. The attest and appraise ASPs are the same as in previous patterns, and running two phrases simultaneously uses the same strategy as for the Cached Certificate Style. The

ease with which our existing core attestation components and supporting libraries made this unique scenario executable speaks to their general-purpose utility.

7.3.6 Layered Background Check

The final Flexible Mechanisms pattern is the Layered Background Check depicted in Figure 7.16, with corresponding Copland phrase in Figure 7.15.

```
*P0,n: @P1[((attest P1 sys) ->
              (attest P3 att) ->
                (attest P4 att)
            +~+
            (@P3[(attest P3 sys)]
               +~+
              @P4[(attest P4 sys)])) ->
           @P2[(appraise P2 it) -> !]]
```

Figure 7.15. Copland phrase for Layered Background Check.

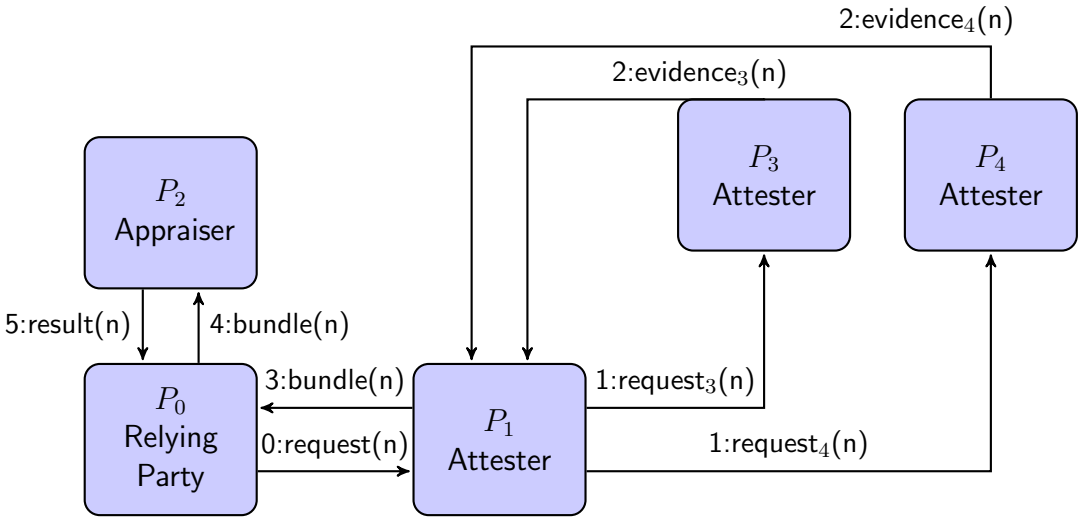

Figure 7.16. Layered background check. Fig 9 on pg. 29:19 of [27].

This scenario involves one top-level client, the Relying Party at place P_0. The initial request bound to a nonce n is sent to place P_1, which then orchestrates the remaining local attestations and requests. P_1 first performs a collection of direct and indirect attestations simultaneously. The set of direct attestations are of itself, P_3, and P_4, in that order. The indirect measurements are requests to P_3 and P_4 to attest themselves, and those requests are free to occur in any order. After all of these attestations complete, P_1 sends a final request to the Appraiser at P_2 to perform the appraise ASP and sign the result of appraisal.

By the time evidence arrives at P_2 for appraisal, the evidence bundle is quite complex. Part of this complexity comes from the nested sub-attestations encoded as results of the attest ASPs at the various places. But even the top-level evidence sequence itself has a non-trivial amount of structure. Luckily we can leverage the evidence semantics of Copland to help here. However the pipeline of ASPs in this phrase differs from previous patterns where appraise only followed a single attest. In this case we have to provide appraise with more structural information to interpret the more sophisticated evidence type. To accomplish this we configure the top-level appraise ASP statically with the *prefix* of the overall phrase before appraisal, as in Figure 7.17. To configure appraise we chose to encode the above phrase prefix into a String parameter of the ASP argument list. This kept the implementation of the appraise ASP as general as possible, and allowed static configuration before attestation launch time.

The final wrinkle here is the generalized appraisal procedure still needs to know how to appraise each individual attest ASP. Recall that the procedure will call checkASP for each raw evidence result of attest. But because attest constructs and encodes an AttestResult, we can implement appraise such that it decodes and

```
@P1[((attest P1 sys) ->
               (attest P3 att) ->
                 (attest P4 att)
             +~+
             (@P3[(attest P3 sys)]
                +~+
               @P4[(attest P4 sys)]))]
```

Figure 7.17. Non-appraisal prefix of the Copland phrase for the Layered Background Check pattern.

interprets that structure, extracts the phrase selected by attest, and use it to compute the expected evidence shape for that sub-attestation as input to generalized appraisal. Notice this is a similar appraisal strategy as in the Certificate Style pattern above, but more principled. Before we relied on the fact that a single attest ASP preceded appraise, so it was easy to guess where the appropriate AttestResult structure would be. The new strategy supports handling arbitrary embedding of attest ASP results within the overall evidence structure.

Chapter 8

Conclusion and Future Work

In this work we have presented a collection of core components that aid in the design, execution, and analysis of layered attestation protocols. The Copland Virtual Machine provides a fine-grained execution and bundling semantics for Copland attestations that is formally verified to respect the Copland reference semantics for events and evidence shapes. A dual generalized appraisal procedure unbundles and checks evidence integrity, and is formally verified to achieve appraisal coverage – everything attested is appraised with sufficient cryptographic strength. We lift the formal properties of these artifacts into the Copland Verification Architecture to discharge assumptions about honest Copland participants and compose with an analysis of protocol strength in the presence of a capable adversary. This end-to-end workflow is exercised to aid in the iterative design and analysis of a UAV demonstration platform instrumented with attestation. The analysis incorporates custom first-order encodings of the Security Architecture and measurement dependencies on the ground station to justify mitigation of attack models. Finally we instantiate the diverse Flexible Mechanisms benchmark of attestation patterns within the Haskell AM prototype. Implementing these pat-

terns required concrete instantiation of primitive attestation services and careful integration with the formal artifacts generated via code synthesis.

Formal verification was vital to confirm key properties of the intricate execution and bundling semantics of the CVM and appraisal components, including precise parameters passed to external services. Rigid specifications of correctness uncovered simple but subtle bugs like parameter swapping early in implementation of the CVM. Later on it highlighted less obvious corner cases in the appraisal procedure where specific evidence structures eluded appraisal. The interplay of verification and system prototyping brought insights that make the attestation mechanisms more robust and inform the design of future systems. Refining the reference semantics to a concrete Haskell implementation exposed complexity at each stage, including the subtle differences in evidence bundling among primitive ASP terms, and strategies for parallel execution of Copland phrases. Implementing the Flexible Mechanisms patterns uncovered complexity of data flow in ASP evidence pipelines, highlighting the need for constrained disclosure of evidence to certain ASPs. The prototyping effort forced careful integration of the core formal components with non-formal services in the surrounding Haskell environment.

The contributions of this work provide a system designer with a set of building blocks that aid in a principled understanding of where risk lies on a system. Rather than a once-and-for-all theorem that implies absolute security, formal properties of these attestation components add rigor and inform the context of a larger system-level security argument. Ongoing work that places the CVM and appraisal components within a larger attestation stack appears in Figure 8.1. Above protocol execution is a negotiation phase that selects a protocol amenable to all participants [19]. Below is a synthesis phase that extracts the CVM and its security

architecture to a CakeML program [4, 31] and seL4 component specification that both enjoy formal semantics down to metal. Viewed another way, these phases will support automatic synthesis of three distinct but related artifacts: *protocol, architecture, and code*. The core components defined and formally verified in this work serve as the fundamental link in this end-to-end attestation stack.

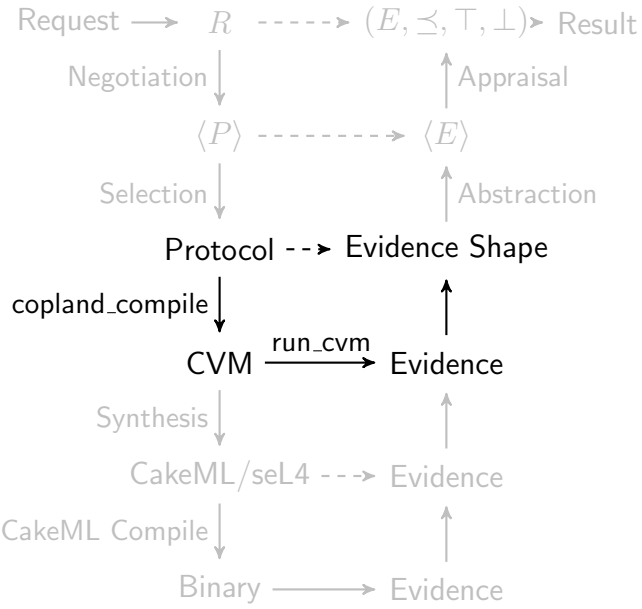

Figure 8.1. Verification stack showing verification dependencies and execution path. Solid lines represent implementations while dashed lines represent mathematical definitions.

Chapter 9

Related Work

Remote Attestation has its roots in the related fields of trusted computing, system-level security, and protocol analysis. There are also a number of academic and industrial tools that leverage remote attestation to gain varying levels of trust in systems. With that in mind, the goal of the current work is not to replace these approaches and tools. Rather, we aim to develop a framework that is general enough to orchestrate these existing tools as specialist services, then combine and report their evidence in a principled way. This will support increasingly diverse attestation scenarios that a single tool/approach cannot support on its own. These new scenarios may involve multiple participants, each with diverse system architectures, cryptographic, and measurement capabilities.

Although our framework is designed to bundle existing attestation services, it is also an attestation service itself. Therefore, it incorporates design principles known to be sound from prior attestation systems. In particular, the designs of the Copland language and CVM attestation execution semantics are careful to support properties like bottom-up measurement [60], robust evidence bundling [59], constrained disclosure and trustworthy mechanism [12].

9.1 Models of Security

Our remote attestation framework is designed with the assumption that attackers can and will corrupt systems. In their early analysis of cryptographic communication protocols, Dolev and Yao present a precise model of an adversary that can eavesdrop on all messages exchanged [15]. They make very few assumptions about the adversary; its only limitations are cryptographic laws of the messages involved. Although attacker capabilities are not the primary focus of this work, we do design our framework with a powerful adversary in mind that can view all messages exchanged between protocol participants. Another influence on our design is the notion that security is not absolute, and that systems will inevitably fail [40]. It is therefore important that our remote attestation framework provides ways to incrementally constrain the adversary with effective measurement strategies, making their attempts to corrupt the system and go undetected as difficult as possible. From this perspective, the primary utility of our formally verified core attestation components is thus their *isolation of risk* on layered systems rather than guarantees of absolute security.

9.2 Formally Verified System-level components

The aggregation of high-assurance components creates exciting opportunities for building system-level security arguments. Below we list a number of core components to system-level security have come to maturity in recent years with the help of formal verification and mechanized proof. However, the security properties gained from these verified implementations are of little use if an adversary corrupts layers below them or corrupts adjacent components that they rely on for successful

execution. This highlights the importance of our remote attestation infrastructure, not only to attest the authenticity of components like these on a system, but to build the attestation infrastructure itself.

- seL4 [33]: seL4 is an operating system kernel with strong formal guarantees of component isolation and access control. High-level security properties like non-interference are specified and verified in HOL4, then refined to an efficient, semantically-equivalent C implementation. Isolation mechanisms like seL4 are critical for layered attestation systems to protect measurement components from their potentially untrusted targets.

- Compcert [37]: Compcert is a C compiler with formal guarantees in Coq of semantics-preservation down to various mainstream machine code targets. Attestation components like cryptographic and measurement implementations are written in C due to their low level nature. Compcert ensures bugs are not introduced into these critical components during compilation.

- CakeML [36]: CakeML is an ML-like language with a compiler to machine code formally verified in HOL4. Ongoing work [4,31] will synthesize CakeML implementations of attestation components from their Coq specifications.

- VellVM [71]: VellVM is a framework for reasoning in Coq about programs expressed in LLVM's intermediate representation. Its uses include hardening C programs against spatial memory safety violations, along with other formally verified transformations with negligible performance impact.

- FSCQ [9]: FSCQ is a formally certified crash-proof file system implemented and verified in Coq. Its proofs ensure a file system will recover its contents correctly under arbitrary sequences of system crashes and reboots.

- Kami [10]: Kami is a framework in Coq for implementing, specifying, verifying Bluespec-style hardware components. It supports modular hardware verification and synthesis of high-level designs into low-level circuits.

- CertiKOS [25]: CertiKOS is a certified secure OS Kernel specified and verified in Coq. CertiKOS is a general-purpose OS kernel with concurrent features like fine-grained locking that achieves functional correctness of system calls akin to seL4, but focuses less on high-level security properties.

- SELinux [39]: SELinux is a mandatory access control policy enforcement architecture for Linux systems. While microkernels like seL4 provide separation guarantees for components at the system-level, SELinux supports more fine-grained policies for application-level interactions within Linux.

- F* [64]: F* is a functional language in the ML family that has support for dependent types in the form of refinement type annotations on code, alongside push-button automation via linkage to SMT solvers. Its support for monadic effects, systems language embeddings (Low*), and synthesis capabilities make it an accessible environment for low-level formal developments. Its verified implementations include a growing list of fundamental cryptographic and distributed systems algorithms.

- Cryptol [7]: Cryptol is a domain-specific language and framework for specifying cryptographic algorithms. While verified compilers guarantee the semantics of cryptographic implementations are preserved to metal, Cryptol ensures functional correctness of the algorithms themselves with respect to mathematical specifications of cryptographic strength.

9.3 Trusted Platform Module(TPM)

The Trusted Platform Module (TPM) [67] is a cryptographic co-processor designed by the Trusted Computing Group (TCG) to reliably store and report evidence of system integrity. Although we don't rely on the TPM explicitly in this work, our attestation framework was designed to be configurable with a hardware root of trust that has the properties similar to a TPM. It is a passive device with access-controlled storage locations called Platform Configuration Registers (PCRs) that are populated with hash measurements. The only way to update a PCR is by extending its previous hash value, and nonresettable PCRs ensure that a complete history of measurements are preserved, starting from the earliest code that executes on a system. Each TPM provides a unique identity by means of a private Endorsement Key (EK) which is burned into the chip during manufacturing. The EK derives and protects another key called the Storage Root Key (SRK), which in turn can derive a collection of keys used to encrypt and store data outside of the TPM, only to be decrypted by way of the TPM. The EK can also generate Attestation Keys(AKs) that serve as aliases for the EK when reporting the contents of its PCRs in a TPM Quote. Signing quotes with the AK allows a platform to prove it has a manufacturer-approved TPM, but also preserve some of its anonymity by using multiple AKs. TPMs can serve as a hardware root of trust for many important remote attestation scenarios. However, care must still be taken to avoid using them in unsound ways [60].

9.4 Process Calculi

A distinguishing feature of our Remote Attestation framework is that it is language-based. In particular, we allow specification of attestation patterns as language terms called *phrases* in the Copland framework. The syntax of Copland phrases is reminiscent of earlier process calculi like Milner's Calculus of Communicating Systems CCS [45] and Hoare's Communicating Sequential Processes [29] in its ability to specify sequential and parallel composition of subterms. Parallel subterms in Copland do not communicate implicitly like they might in these process calculi. Instead, Copland uses an @ term to specify an explicit communication channel, specialized for sending attestation requests and receiving evidence responses. The notion of explicit handling of channels first appeared in a descendant of CCS called the pi-calculus [46]. An even more recent extension of the pi-calculus, called the spi-calculus [1], adds cryptographic primitives to the pi-calculus. This is similar to primitive terms in Copland like SIG and HSH that perform cryptographic operations over accumulated evidence.

9.5 Remote Attestation Frameworks

9.5.1 IMA

IMA(Integrity Measurement Architecture) [62] is a tool that checks the integrity of all executable content loaded onto a Linux system. It uses a "measure-before-execute" strategy, placing hashes of all executables into TPM (Trusted Platform Module) protected storage before loading and executing them. A remote system can then request a composite TPM quote of these hash values to asses the integrity of the target system. IMA is a baseline system measurement to

detect attacks such as OS rootkits that attempt to initialize a system in a malicious state. In this sense, it can be viewed as a "boot-time" or "load-time" measurement that guarantees the system starts in some expected state immediately after the code is loaded. It also provides modest run-time guarantees by instrumenting the Linux kernel to measure any executables loaded after boot. IMA supports a form of layered attestation, albeit a fixed and shallow one: the BIOS measures the integrity of the core kernel code, then the instrumented kernel code measures the integrity of changes to itself (modules loaded, user level processes started), and measures executable code loaded into its processes at run-time using file_mmap LSM hooks to "induce a measurement on any file before it is mapped executable into virtual memory" [62].

The IMA measurement strategy and the corresponding system architecture must be fixed prior to fielding a system. While the measurement list stored in-kernel records and reports the order executable components start, it does not enforce, or allow the user to prescribe, the order in which components start after boot. In our attestation framework, we could invoke IMA to get a quote as a baseline measurement of the system, ensuring the proper components at least started in an expected state (correct executable loaded, proper configuration files used). A quote from [62] that summarizes the utility of IMA is as follows: "...our architecture ensures that the breadth of the system is measured ... but the depth of measurement ... is not complete, but it is extensible, such that further measurements to increase confidence in integrity are possible".

PRIMA (Policy-Reduced IMA) [30] is an extension to IMA that is less coarse-grained and leverages access control policies on the target platform to eliminate unnecessary measurements and allows untrusted code to run alongside trusted

code on a system. Their approach uses SELinux MAC policies to enforce information flow between processes, then uses these policies to inform measurement decisions during attestation. This limits re-measurement, and reduces false positives because a failed integrity check of a non-dependency of the target will not affect its integrity. PRIMA can be viewed as an optimization of IMA, and still provides a fixed, baseline measurement of Linux systems.

9.5.2 DR@FT

DR@FT (Dynamic Remote Attestation Framework and Tactics) [70] is similar to PRIMA, in that it uses access control policies to optimize measurement. In addition, they focus on the most recent changes to the system. They also introduce a graph-based integrity violation analysis that ranks policy violations and give an appraiser quantitative guidance to determine the severity of violations and how efficiently they might be resolved.

9.5.3 Bind

BIND [63] provides a more fine-grained notion of attestation: it measures, starts, and protects individual components, rather than performing system-wide measurement. It also incorporates attestation of input data so that an attestation can produce a chain of evidence about how data is transformed, perhaps by multiple components on a system. This transitive chain of data transformations can be checked by a single signature verification. BIND makes appraisal easier since it is not sensitive to small changes of configuration on the target when those changes are not relevant to the code of interest. It also moves beyond the load-time guarantees of traditional TCG approaches like IMA by tracking and appraising data

transformed by processes at run-time.

9.5.4 Policy Driven Remote Attestation

The Policy Driven Remote Attestation framework [24] leverages data access policies to inform the requirements of remote attestation. The attestation process itself is not new: they invoke IMA as-is. However, they introduce an authorization logic and an accompanying architecture that informs the parameters of attestation. An interesting part of their architecture is the Attestation Authority that manages a database of measurement values. This isolates to a dedicated component the complex management and often-changing "golden hashes" of different software versions and patches.

9.5.5 Copilot

Copilot [49] is a kernel-integrity monitor that extends beyond boot and load-time, supporting arbitrary checks on main memory at run-time. While this is coarse-grained, it is also flexible, and at the time of writing detected 12 real-world rootkit attacks on Linux. It aims to detect the most sophisticated types of rootkit attacks that modify the OS kernel on an already-compromised host. Their prototype checks MD5 hashes of the kernel's code, loaded kernel modules, and some of the kernel's critical data structures. Since their implementation is PCI-based, their view of the host is limited to main memory (they are unable to pause the host's CPU execution or examine its registers). This limits the precludes the guarantee that no malicious code was executed, but does limit the attacker options to "timing attacks and extremely advanced relocation attacks" [49].

9.5.6 TrustLite and TyTan

TrustLite [35] is a hardware security architecture that supports remote attestation on embedded devices. They modify MPU and CPU execution engine hardware to provide isolation of software components on a system. An advantage of their architecture is that it is general enough to be instantiated on different operating systems and software stacks. The EA-MPU (Execution-AWARE MPU) and secure exception engine protects against untrusted exception handlers, and their Secure Loader configures their architecture and can serve as a root of trust in remote attestation scenarios. TyTan [6] builds on the TrustLite architecture, extending it to provide execution-time guarantees for applications with real-time requirements.

9.6 Analysis of Remote Attestation

9.6.1 Principles of Remote Attestation

In [12], Coker et. al define a list of design principles that should guide the development of attestation systems. These principles play a central role in the design of Copland, and also in the attestation manager framework presented in this work. The five principles are: fresh information, comprehensive information, constrained disclosure, semantic explicitness, and trustworthy mechanism. *Fresh information* means that measurements should reflect the current state of the running target as much as possible. In our framework, our explicit nonce generation and management in the AM Monad is towards this goal of freshness. *Comprehensive information* means that measurement tools on a target system should be capable of delivering a comprehensive description of system state to a

remote appraiser. This is handled in Copland as the ability to specify arbitrary system measurements as Attestation Service Providers, and in the CVM Monad that faithfully carries out the entire list of specified measurements.

The next principle, *constrained disclosure*, is often at odds with complete information. It says that the target of attestation has the right to refuse to disclose certain aspects of its configuration to actors it does not trust with this information. While we do not address this principle explicitly in our attestation framework, we rather assume the existence of a policy and negotiation phase responsible for rejecting Copland phrases that violate privacy concerns of the target. *Semantic explicitness* says that the content of attestation should have a well-defined, logical form such that an appraiser can compose multiple attestations of the target to make a security decision. This principle is the single-most driving force behind the Copland effort. Copland phrases make explicit the appraiser's goals during attestation, and Copland evidence ensures the results are bundled in a way that is sound and predictable. Finally, *trustworthy mechanism*, states that attestation evidence should be accompanied by meta-evidence indicating the trustworthiness of the attestation mechanisms themselves. The primary goal of the current work is to ensure this principle by formally verifying key components of an attestation infrastructure. An appraiser can then rely on knowledge of the specific attestation services invoked, and know that these services were coordinated faithfully.

9.6.2 Confining adversary actions via measurement

There is a pervasive notion in the attestation literature that a "bottom-up" measurement strategy, where chains of measurements start with a hardware root and build outward, is the most effective strategy to gain trust in systems. Rowe's

work [59] is the first to rigorously formalize this intuition by modeling arbitrary combinations of system measurement events alongside adversary actions that corrupt and repair system components. The model is a graphical representation that supports reasoning about attestation strategies to measure layered systems. The main result of the paper confirms the intuition that bottom-up measurement forces the adversary to perform a "recent or deep" attack on the system to achieve its goals and go undetected. These types of attacks place more burden on the attacker by requiring they corrupt a deeper, more protected component or provide a smaller time window for the attacker to operate. The graphical model of measurement and adversary events is very much akin to Event Systems in the Copland framework. One of the goals of Copland is to leverage this analysis to rank and select Copland phrases that would properly constrain the adversary. The goal of the current work is to ensure that measurement strategies specified in Copland are faithfully executed by the attestation manager, ensuring the graphical analysis on Event Systems remains sound.

9.6.3 Bundling evidence for layered attestation

Since an appraiser must make a trust decision based solely on the evidence it receives from a remote system, the evidence must be bundled in a way that implies properties of how it was collected. Rowe's work that explores bundling strategies for layered attestation [60] compliments his work of confining adversary actions [59] by incorporating a novel theory of evidence structures for layered measurement systems. To accompany the formal model of system events from the accompanying work [59], he adds measurement storage and reporting primitives based on capabilities of the TPM, including PCR extension and quoting. This

supports proving the efficacy of different bundling strategies based on how well they imply proper measurement orderings to constrain the adversary.

After demonstrating some common pitfalls in bundling strategies, he proposes a particular strategy that gives measurement components at each layer exclusive access to their own PCR, but also incorporates evidence from lower layers into higher layers. He then uses his theoretical framework to prove that evidence from this strategy forces a recent or deep corruption from an adversary. Although there is no formal connection from this work to the Copland framework, one could imagine prescribing a bundling strategy as a carefully crafted Copland phrase that interacted with a TPM (or something like a TPM). As it relates to the work in this proposal, a formally verified attestation manager can be shown to perform measurements in a prescribed sequence, somewhat eliminating the need to prove this ordering by the structure of evidence alone. However, there may be portions of an attestation scenario where an attestation manager cannot collect the evidence directly (like early boot), and where properly bundled evidence is critical. This work lays a theoretical foundation for further exploration of this trade-off: balance between analysis of the evidence structure and trust of the evidence collection process.

9.6.4 A Minimalist Approach to Remote Attestation

In work that derives a set of minimal properties necessary for remote attestation [17], the authors begin with a logical description of the security of remote attestation protocols as a game between a appraiser and attester (the authors use the equivalent terms "challenger" and "prover" to mean appraiser and attester). A system is secure if there is a negligible probability of a compromised prover

producing a claim accepted by the challenger as valid. They follow with some assumptions about the entities involved in attestation. First, that the prover is a low-end embedded device with a single thread of control and limited resources. They make few assumptions about the challenger and the adversary, except that the adversary cannot break axioms imposed by hardware like writing to ROM. Next, they derive a set of properties they claim are necessary and sufficient to achieve the security goals under their assumptions. These include exclusive access to the attestation key K, no leaks of K, and immutability, uninterruptibility, and controlled invocation of the main attestation code, such that it must only be "invoked from its intended entry point" [17] . They claim, but do not prove, that those five properties imply security of attestation as they define it. They also claim, but do not describe in detail, how removal of any one of these properties results in a failure of their attestation goals. The paper ends with a brief description of *features* that would satisfy the five properties on a concrete embedded system. Although lacking some specificity, the paper presents a well thought out and organized collection of properties that remote attestation on an embedded platform should reasonably follow.

9.6.5 Negotiation of Attestation Protocols

Prior to execution of an attestation protocol, the involved parties must agree on how they will interact and what information they are willing to share. Toward this goal, Kline [34] proposes a multi-stage communication protocol where the participants attempt to arrive at an attestation protocol amenable to all. The negotiation process is formally specified and verified in Coq with respect to properties like deadlock-freedom and privacy of the involved parties. Sessions are

limited to two participants at a time, and the attestation protocol being derived is a custom collection of measurements desired by each participant of the other. In recent work, Fritz [19] presents a strategy for protocol selection that informs negotiation of Copland-based protocols. This strategy leverages dependent types in Coq and simplified versions of Copland and its evidence structure to generate protocols that meet local privacy policies of individual participants. Ongoing work [18] will extend negotiation to the full Copland language and link it to the formal semantics of attestation and appraisal components presented in this work.

9.7 Formal Verification of Remote Attestation

9.7.1 HYDRA

HYDRA [16] is a design and implementation of a hybrid hardware/software remote attestation platform built upon the formally verified seL4 microkernel. The authors claim that it is the first design to build upon formally verified software components to achieve the goals laid out in a subset of their author's prior work [17]. They leverage seL4 [33] guarantees of memory isolation and access control to protect key components of the attestation infrastructure including the attestation key and the attestation process. They assume a fixed architecture, with one main attestation process called "PR_Att" that is responsible for starting all other userspace processes, performs all measurements, and has exclusive access to the symmetric signing key K. PR_Att listens for requests from a remote appraiser that indicate the process of interest, and the memory range within that process to measure. Copland phrases, in contrast, were designed so that many distinct measurers could cooperate to jointly measure a target, perhaps with measurement agents at different layers in the architecture. Although their minimal

architecture seems to support less exotic attestation scenarios, it allows them to build a convincing security argument. They achieve many security goals we aim for in our infrastructure including key isolation, attestation process protected from untrusted components, trusted boot chain, and measurement freshness. Similar to their design, the security of our attestation manager depends critically on access control properties like those provided by seL4 to protect the core attestation components.

9.7.2 ERASMUS

ERASMUS (Efficient Remote Attestation via Self-Measurement for Unattended Settings) [8] uses HYDRA [16] as the base security architecture, but adds support for "self-measurement" to account for potentially expensive "on-demand" requests in traditional attestation scenarios that may drain critical resources on real-time devices. Self-measurement allows the target to periodically record its own software state, which may be cached and collected by a potentially untrusted verifier. They evaluate ERASMUS using a novel metric called Quality of Attestation (QOA) that is a combination of how a target is attested, how often its state is measured, and how often these measurements are verified. Caching of measurements is one use-case the Copland framework was envisioned to support, so ideas from ERASMUS may prove useful in future work when we experiment with caching strategies.

9.7.3 VRASED

VRASED (Verifiable Remote Attestation for Simple Embedded Devices) [47] is a formally verified remote attestation scheme for low-end embedded devices. The

authors claim it is the first such RA scheme, and the first formal verification "of a HW/SW co-design implementation of any security service". They specify end-to-end RA security and soundness properties in LTL (Linear Temporal Logic), then prove these properties by decomposing verification tasks to hardware and software submodules. The hardware components are specified in Verilog, then automatically translated to a model checker using the Verilog2SMV tool to check their LTL properties. For the software components, they rely on the formally verified cryptographic library HACL*. To incorporate the software components into the end-to-end verification, they manually convert the functional correctness guarantees of the HACL* implementation into an LTL specification, then link it with the LTL specifications of the hardware modules. Their specifications of security and soundness of RA come from their earlier work [17] that defines a necessary and sufficient set of properties to achieve secure RA. Their approach of verifying core properties of their attestation functionality in hardware, but also incorporating the external verification of the HACL* crypto implementation, is compatible with the design goals of the Copland effort. Rather than prove properties about specific hardware, Copland relies on a core, verified attestation manager that coordinates independently-verified components like HACL* as Attestation Service Providers. Although their end-to-end security guarantees are impressive, our attestation managers are designed to support a wide range of attestation scenarios on diverse platforms, rather than the fixed, embedded platform they focus on in this work.

9.8 Measurement Tools

9.8.1 Maat

Maat [48] is a prototype measurement and attestation (M & A) framework designed as a centralized service to coordinate and protect the components involved in attestation protocols. Its functionality includes selecting, collecting, and evaluating integrity measurements that indicate the trustworthiness of a system's static hardware/software configuration and its run-time state. Maat's centralized and componentized design has a number of benefits: it avoids duplication of measurement, supports controlled registration and protection of components, and efficient protocol negotiation/selection. Much of the terminology in the Copland effort is borrowed from the Maat work, including Attestation Manager, Attestation Service Provider, measurement agent, evidence, and target. The Copland language was originally designed with Maat measurement specifications in mind as targets. Our current work can be viewed as a formally verified alternative to Maat. We also envision a use-case, left for future work, where our formally verified Attestation Managers provide a principled means to orchestrate primitive measurers running as Maat services. Maat is currently in the public release process to become open source.

9.8.2 LKIM

LKIM (Linux Kernel Integrity Measurer) [41] is a dynamic measurement tool that observes the runtime state of a Linux kernel. It aims to detect modifications to the kernel such as "kernel-level rootkits" that are notoriously difficult to monitor from components that themselves rely on the potentially compromised kernel. To run outside of the kernel's control, LKIM leverages "virtual-machine introspec-

tion" to view and report the kernel's contents to a remote decision maker. LKIM is an example of an integrity measurer whose evidence cannot be appraised by a single, known golden value, but rather detects anomalous patterns that bring kernel memory outside its behavioral specification. Cross-domain measurement agents like LKIM are critical for supporting more complex, layered attestations: OS kernels act as critical execution contexts for other primitive measurement and attestation components. LKIM remains proprietary, but has been exercised on a variety of critical government applications.

9.8.3 MSRR

MeaSeReR (MSRR) [20] is a general-purpose framework that supports building application-specific dynamic measurers. It extends the GNU Debugger (GDB) to support measurement capabilities that are critical for monitoring the run-time state of a program. These capabilities include sampling code, globals, heap, stack locals, and the call-stack. The framework also provides a measurement policy language to specify complex measurements that are recurring, triggered by system events, or depend on the results of previous measurements. This gives the user a high level interface to quickly develop one-off measurers that are sensitive to the behavior of specific applications. For time-expensive measurements that would stall an application if performed directly, a snapshot feature allows measuring a forked copy of the application offline while the original continues executing. The commands can be invoked either locally or remotely through a JSON RPC, an essential feature for remote attestation scenarios. A performance analysis shows that MSRR has an acceptable effect on normal program execution time benchmarks. The authors conclude with a case study showing how MSSR measurement

policies can detect run-time attacks on DreamChess, an open source chess game for Windows, Mac, and Linux.

9.8.4 Runtime State Verification on Resource-Constrained Platforms

In Runtime State Verification on Resource-Constrained Platforms [11], the authors explore the challenges of implementing runtime measurement agents on resource-constrained platforms such as IoT devices. The challenges presented include a lack of trusted hardware, memory, storage, and power constraints, lack of memory separation, and a diversity of platform services and communication capabilities. They built their prototype on ARM's mbedOS and uVisor micro hypervisor, on the Freescale FRDM-K64F hardware platform, although they argue that their framework is designed to avoid platform-specific features. They limit their measurements to those that can be "comprehensible and verifiable" by an external appraiser, restricting evidence checks to small ranges of values. They chose types of measurers to get as complete a picture of the runtime state as possible given the limited resources.

These measurement types include a static hash of persistent flash memory, runtime analysis of the interrupt vector table, runtime memory introspection of uVisor, and runtime application introspection. They intend that developers on other IoT-like devices will use these measurers as templates, and use their API to build their own platform and application-specific measurers. They do some execution time and power analysis to show that their measurements are largely negligible to performance on their prototype system. Their discussion about matching measurers to well-defined appraisers, and their intent to support integration of platform-specific measurement agents aligns with the design goals of the Copland

framework. They also highlight the diversity of platforms that may be involved in attestation–inspiring generality in our framework to support measurement of such platforms.

Appendix A

Copland + JSON

$$
\begin{array}{rcl}
t & \leftarrow & A \mid @_p\, t \mid (t \rightarrow t) \mid (t \overset{\pi}{\prec} t) \mid (t \overset{\pi}{\sim} t) \\
A & \leftarrow & \mathsf{ASP}\ \bar{a} \mid \mathsf{CPY} \mid \mathsf{SIG} \mid \mathsf{HSH}
\end{array}
$$

Figure A.1. Copland Phrase grammar where:
$\bar{a} = (m,\ \bar{s},\ p,\ r)$; $m = \mathit{asp\_id} \in \mathbb{N}$; $\bar{s}$ is a list of string arguments; $p = \mathit{place\_id} \in \mathbb{N}$; $r = \mathit{target\_id} \in \mathbb{N}$; and $\pi = (\pi_1, \pi_2)$ is a pair of evidence splitting functions.

$$
\begin{array}{rcl}
E_T & \leftarrow & \mathsf{mt} \mid \mathsf{N_E}\ \mathsf{n} \mid \mathsf{ASP_E}\ \bar{a}\ p\ E_T \\
 & & \mid \mathsf{SIG_E}\ p\ E_T \mid \mathsf{HSH_E}\ p\ E_T \\
 & & \mid \mathsf{SS_E}\ E_T\ E_T \mid \mathsf{PP_E}\ E_T\ E_T
\end{array}
$$

Figure A.2. Evidence Type grammar where:
$\bar{a}$ and p are as in Fig. 3.1 and $n = \mathit{nonce\_id} \in \mathbb{N}$.

$$
\begin{array}{rcl}
E_{Tc} & \leftarrow & \mathsf{mt}_c \mid \mathsf{N_c}\ \mathsf{n}\ \mathsf{bs} \mid \mathsf{ASP_c}\ \bar{a}\ p\ \mathsf{bs}\ E_{Tc} \\
 & & \mid \mathsf{SIG_c}\ p\ \mathsf{bs}\ E_{Tc} \mid \mathsf{HSH_c}\ p\ \mathsf{bs}\ E_T \\
 & & \mid \mathsf{SS_E}\ E_{Tc}\ E_{Tc} \mid \mathsf{PP_E}\ E_{Tc}\ E_{Tc}
\end{array}
$$

Figure A.3. Typed Concrete Evidence grammar where:
$\bar{a}$, p, and n are as in Fig. 3.5 and $\mathsf{bs} \in \mathsf{BS}$ (binary values).

A.1 General ADT JSON Schema

We represent Copland terms (t in Figure A.1), Evidence Types (E_T in Figure A.2), and Typed Concrete Evidence (E_{Tc} in Figure A.3) as Algebraic Data Types (ADTs) in the Haskell AM prototype. In JSON we represent each ADT as an object with two fields:

1. constructor-the constructor name as a JSON string (< string >). Constructor names must be unique for unambiguous parsing.

2. data-An ***ordered*** JSON array (< array >) that holds the arguments for that particular constructor. Members of the data array will differ from constructor to constructor.

The general schema for ADTs (labeled by placeholder < ADT >) is as follows:

```
{
  "constructor": < string >,
  "data": < array > | < ADT >
}
```

The < ADT > alternative for the data field accounts for degenerate nesting of constructors (for example in the ASP constructors below).

A.2 Copland Phrase JSON Schemas

The following JSON object schemas correspond to the constructors of the Copland phrase grammar (t) in Figure A.1, and satisfy the < term > placeholder. The < number > and < string > placeholders are for the standard json number

and string datatypes. The order of items in the "data" subarray is significant–they match the order of arguments to each constructor. Finally, the placeholder $<$ "ALL''|"$NONE''$ $>$ stands for a JSON string that is either the constant "ALL" or "NONE".

$<$ asp_params $>$:= [$<$ number $>$, [$<$ string $>$], $<$ number $>$, $<$ number $>$]

```
{
  "constructor": "Coq_asp",
  "data": {"constructor": "ASPC",
           "data": < asp_params >
           }
}
```

```
{
  "constructor": "Coq_asp",
  "data": {"constructor": "SIG"}
}
```

```
{
  "constructor": "Coq_att",
  "data": [ < number >,
            < term > ]
}
```

< SP > := “ALL” | “NONE”

```
{
  "constructor": "Coq_bseq",
  "data": [ [< SP >, < SP >],
            < term >,
            < term > ]
}
```

A.3 Copland Evidence Type Schemas

The following JSON object schemas correspond to the constructors of the Evidence Type grammar (E_T) in Figure A.2, and satisfy the < evidence_type > placeholder.

```
{
  "constructor": "Coq_mt"
}
```

```
{
  "constructor": "Coq_uu",
  "data": [ < asp_params >,
            < number >,
            < evidence_type > ]
}
```

```
{
  "constructor": "Coq_gg",
  "data": [ < number >,
            < evidence_type > ]
}
```

```
{
  "constructor": "Coq_ss",
  "data": [ < evidence_type >,
            < evidence_type > ]
}
```

A.4 Copland Typed Concrete Evidence Schemas

The following JSON object schemas correspond to the constructors of the Typed Concrete Evidence grammar (E_{Tc}) in Figure A.3, and satisfy the < evidence_conc > placeholder. Note: Constructor fields that hold binary data (bs parameters in the grammar) become base64-encoded JSON strings (< b64_string >), holding arbitrary binary data–hashes, nonces, signatures, etc:

< b64_string > := < string > (Base64 encoded)

```
{
  "constructor": "Coq_nnc",
  "data": [ < number >, < b64_string > ]
}
```

```
{
  "constructor": "Coq_uuc",
  "data": [ < asp_params >,
            < number >,
            < b64_string >,
            < evidence_conc > ]
}
```

```
{
  "constructor": "Coq_hhc",
  "data": [ < number >,
            < b64_string >
            < evidence_type > ]
}
```

```
{
  "constructor": "Coq_ssc",
  "data": [ < evidence_conc >,
            < evidence_conc > ]
}
```

A.5 Message Schemas

$$
\begin{aligned}
RequestMessage = \{ & \\
toPlace &:: \mathsf{p}, \\
fromPlace &:: \mathsf{p}, \\
reqNameMap &:: \mathsf{p} \Rightarrow \mathsf{Address}, \\
reqTerm &:: \mathsf{t}, \\
reqEv &:: [\ \mathsf{bs}\ ] \}
\end{aligned}
$$

$$
\begin{aligned}
ResponseMessage = \{ & \\
respToPlace &:: \mathsf{p}, \\
respFromPlace &:: \mathsf{p}, \\
respEv &:: [\ \mathsf{bs}\ ] \}
\end{aligned}
$$

Figure A.4. Request and Response Message record structures.

We represent Request and Response Messages as record structures in Figure A.4. Their respective JSON object schemas are as follows:

```
{
  "toPlace": < number >,
  "fromPlace": < number >,
  "reqNameMap": < nameMap >,
  "reqTerm": < term >,
  "reqEv": < raw_evidence >
}
```

```
{
  "respToPlace": < number >,
  "respFromPlace": < number >,
  "respEv": < raw_evidence >
}
```

< raw_evidence > := [< b64_string >]

< nameMap > is a JSON object of the following form:

$\{pl_1{:}addr_1, pl_2{:}addr_2, ...\}$

where pl_1, pl_2, ... are JSON key strings that represent a Copland place identifier (i.e. "1", "2", ...) and $addr_1$, $addr_2$, ... are JSON strings (< string >) that represent platform addresses. We leave address strings abstract in this specification, but a common usage would be a string of the form ip:port.

$$SigRequestMessage = \{evBits \quad :: \quad \mathsf{bs}\}$$

$$SigResponseMessage = \{sigBits \quad :: \quad \mathsf{bs}\}$$

Figure A.5. Sig Request and Response Message record structures.

We represent Sig Request and Response Messages as record structures in Figure A.5. Their respective JSON object schemas are as follows:

```
{
  "evBits": < b64_string >
}
```

```
{
  "sigBits": < b64_string >
}
```

$$
\begin{aligned}
AspRequestMessage = \{ & \\
aspArgs &:: \bar{a}, \\
aspInputEv &:: [\ bs\ ]\ \}
\end{aligned}
$$

$$AspResponseMessage = \{aspBits :: bs\}$$

Figure A.6. Asp Request and Response Message record structures.

We represent Asp Request and Response Messages as record structures in Figure A.6. Their respective JSON object schemas are as follows:

```
{
  "aspArgs": < asp_params >,
  "aspInputEv": < raw_evidence >
}
```

```
{
  "aspBits": < b64_string >
}
```

References

[1] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Infomration and Computation*, 148:1–70, 1999.

[2] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 3–15, New York, NY, USA, 2008. ACM.

[3] S. Balachandran, C. A. Munoz, M. C. Consiglio, M. A. Feliu, and A. V. Patel. Independent configurable architecture for reliable operation of unmanned systems with distributed onboard services. In *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pages 1–6, 2018.

[4] T. Barclay. Proof-producing synthesis of cakeml from coq. Phd proposal defense, The University of Kansas, Lawrence, KS, April 2022.

[5] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan. Remote attestation procedures architecture. `https://datatracker.ietf.org/doc/draft-ietf-ratsarchitecture/`, 2021.

[6] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: Tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[7] S. Browning. Cryptol, a dsl for cryptographic algorithms. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, New York, NY, USA, 2010. Association for Computing Machinery.

[8] X. Carpent, N. Rattanavipanon, and G. Tsudik. ERASMUS: efficient remote attestation via self- measurement for unattended settings. *CoRR*, abs/1707.09043, 2017.

[9] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[10] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), Aug. 2017.

[11] J. Clemens, R. Pal, and B. Sherrell. Runtime state verification on resource-constrained platforms. In *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, pages 1–6, 2018.

[12] 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.

[13] G. S. Coker, J. D. Guttman, P. A. Loscocco, J. Sheehy, and B. T. Sniffen. Attestation: Evidence and trust. In *Proceedings of the International Conference on Information and Communications Security*, volume LNCS 5308, 2008.

[14] D. Coutts, D. Stewart, et al. Data.bytestring haskell library.

[15] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198 – 208, March 1983.

[16] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. Hydra: Hybrid design for remote attestation (using a formally verified microkernel). In *Proceedings of the 10th ACM*

Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '17, pages 99–110, New York, NY, USA, 2017. Association for Computing Machinery.

[17] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to Remote Attestation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, 2014.

[18] A. Fritz. Personal communication, 2022.

[19] A. Fritz. Type dependent policy language (in press). Ms thesis, The University of Kansas, Lawrence, KS, May 2021.

[20] J. Gevargizian and P. Kulkarni. Msrr: Measurement framework for remote attestation. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 748–753. Dependable, Autonomic and Secure Computing (DASC '18), 2018.

[21] E. Ghassabani, A. Gacek, M. W. Whalen, M. P. E. Heimdahl, and L. Wagner. Proof-based coverage metrics for formal verification. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 194–199. IEEE Press, 2017.

[22] A. Gill. Domain-specific languages and code synthesis using Haskell. *Commun. ACM*, 57(6):42–49, June 2014. Also appeared in ACM Queue, Vol 12(4), April 2014.

[23] D. Gollmann. Why trust is bad for security. *Electronic Notes in Theoretical Computer Science*, 157(3):3 – 9, 2006. Proceedings of the First International Workshop on Security and Trust Management (STM 2005).

[24] A. Gopalan, V. Gowadia, E. Scalavino, and E. Lupu. Policy driven remote attestation. In R. Prasad, K. Farkas, A. U. Schmidt, A. Lioy, G. Russello, and F. L.

Luccio, editors, *Security and Privacy in Mobile Information and Communication Systems*, pages 148–159, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[25] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjoberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, USA, 2016. USENIX Association.

[26] D. Hardin, T. D. Hiratzka, D. R. Johnson, L. Wagner, and M. Whalen. Development of security software: A high assurance methodology. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 266–285, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] S. Helble, I. Kretz, P. Loscocco, J. Ramsdell, P. Rowe, and P. Alexander. Flexible mechanisms for remote attestation. *ACM Transactions on Privacy and Security*, September, 2021.

[28] S. Ho, O. Abrahamsson, R. Kumar, M. O. Myreen, Y. K. Tan, and M. Norrish. Proof-producing synthesis of cakeml with I/O and local state from monadic HOL functions. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference (IJCAR)*, volume 10900 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2018.

[29] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–77, 1978.

[30] T. Jaeger, R. Sailer, and U. Shankar. Prima: Policy-reduced integrity measurement architecture. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*, SACMAT '06, pages 19–28, New York, NY, USA, 2006. Association for Computing Machinery.

[31] G. Jurgensen, A. Petz, P. Alexander, T. Barclay, E. Komp, M. Neises, and A. Cousino. A copland attestation manager (am) in cakeml. github repository.

https://github.com/ku-sldg/am-cakeml, 2021.

[32] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Commununications of the ACM*, 53(6):107–115, 2010.

[33] 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.

[34] P. Kline. Remote attestation protocol verification with a privacy emphasis. Ms thesis, The University of Kansas, Lawrence, KS, July 2018.

[35] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[36] 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.

[37] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[38] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[39] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, USA, 2001. USENIX Association.

[40] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *In Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.

[41] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, STC '07, pages 21–29, New York, NY, USA, 2007. ACM.

[42] S. Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[43] A. Martin et al. The ten page introduction to trusted computing. Technical Report CS-RR-08-11, Oxford University Computing Labratory, Oxford, UK, 2008.

[44] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, Feb 2010.

[45] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[46] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[47] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik. Vrased: A verified hardware/software co-design for remote attestation. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1429–1446, USA, 2019. USENIX Association.

[48] J. A. Pendergrass, S. Helble, J. Clemens, and P. Loscocco. Maat: A platform service for measurement and attestation. *arXiv preprint arXiv:1709.10147*, 2017.

[49] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194. USENIX Association, 2004.

[50] A. Petz. copland-avm, nfm21 release. https://github.com/ku-sldg/copland-avm/releases/tag/v1.0, 2020.

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

[52] 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.

[53] A. Petz, G. Jurgensen, and P. Alexander. Design and formal verification of a copland-based attestation protocol. *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'21)*, Nov 20-22, 2021.

[54] A. Petz and E. Komp. haskell-am. https://github.com/ku-sldg/haskell-am, 2020.

[55] U. PLSE. Verdi. https://github.com/uwplse/verdi, 2016.

[56] 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.

[57] J. D. Ramsdell. Chase: A model finder for finitary geometric logic. https://github.com/ramsdell/chase, 2020.

[58] 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.

[59] P. D. Rowe. Bundling Evidence for Layered Attestation. In *Trust and Trustworthy Computing*, pages 119–139. Springer International Publishing, Cham, Aug. 2016.

[60] P. D. Rowe. Confining adversary actions via measurement. *Third International Workshop on Graphical Models for Security*, pages 150–166, 2016.

[61] J. Rushby. Software verification and system assurance. In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 3–10, 2009.

[62] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementatation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, Berkeley, CA, 2004. USENIX Association.

[63] E. Shi, A. Perrig, and L. Van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Security and Privacy, 2005 IEEE Symposium on*, pages 154–168. IEEE, 2005.

[64] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, Jan. 2016.

[65] H. D. Team. Software transactional memory haskell library.

[66] The Coq Development Team. Coq.

[67] Trusted Computing Group. *TCG TPM Specification*. Trusted Computing Group, 3885 SW 153rd Drive, Beaverton, OR 97006, version 1.2 revision 103 edition, July 2007.

[68] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[69] D. Woos, J. R. Wilcox, K. Simmons, K. Palmskog, and R. Doenges. Structtact coq library. https://github.com/uwplse/StructTact, 2020.

[70] W. Xu, G.-J. Ahn, H. Hu, X. Zhang, and J.-P. Seifert. Dr@ft: Efficient remote attestation framework for dynamic systems. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *Computer Security – ESORICS 2010*, pages 182–198, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[71] J. Zhao, S. Nagarakatte, M. Martin, and S. A. Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL), 2012*, POPL '08, pages 3–15. ACM, 2012.