Design and Formal Verification of a Copland-based Attestation Protocol

Adam Petz*, Grant Jurgensen*, Perry Alexander*

*Information and Telecommunication Technology Center
The University of Kansas
Lawrence, KS 66045
{ampetz,gajurgensen,palexand}@ku.edu

***Index Terms*—Remote Attestation, Formal Methods, Verification**

***Abstract*—We present the design and formal analysis of a remote attestation protocol and accompanying security architecture that generate evidence of trustworthy execution for legacy UAV flight planning software. Our premise is verification of the remote attestation system is far simpler than verification of the flight planning software, and results are reusable in other applications. Effective remote attestation requires that target system measurements run in isolation, in the correct order, and reporting is performed with integrity. For formal guarantees of measurement ordering and cryptographic evidence strength, we leverage the Copland language and Copland Virtual Machine execution semantics. We use the separation properties of the seL4 microkernel to protect keys and isolate attestation mechanisms. The protocol and architecture together serve to discharge assumptions made by an existing higher-level model-finding tool to characterize all ways an active adversary might corrupt the target and go undetected. Finally, we instantiate the Attestation Manager Monad on a platform that is remote to the target, serving as an environment to generate a random nonce, orchestrate Copland requests, and perform sound appraisal over evidence results. By leveraging components that are amenable to formal analysis, we demonstrate a principled way to design an attestation protocol and argue for its end-to-end correctness.**

I. Introduction

A common goal for communicating systems is *trust* where "principal B *trusts* principal A with regard to statement p if and only if from the fact that A has said p, B infers that p was true at a given time" [1]. Establishing trust in A allows B to accept as true what A tells it. We say that B trusts A if it can strongly identify A and either directly observe A behaving as expected or indirectly observe A through a trusted third party. Semantic remote attestation [1, 2] is one mechanism supporting trust establishment. In remote attestation an *appraiser* (B) requests an *attestation* from a remote *target* (A). The target responds by performing the requested attestation and returning *evidence* of its state to the appraiser. The appraiser assesses the evidence and chooses whether to trust the remote system.

This work is funded in part by the NSA Science of Security initiative contract #H98230-18-D-0009 and Defense Advanced Research Project Agency contract #HR0011-18-9-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

This work centers on transforming a legacy system into a system where trust can be established and maintained over time. Remote attestation is added to a UAV control system to establish trust between the UAV and a ground station providing way points for navigation. Formal methods–proof and model finding–are leveraged throughout the retrofit process to ensure correctness in the resulting system. The goal being demonstrating the use of formal methods in system design involving multiple components, requirements, and constraints.

Our work rests on two premises. First, When assessing systems with high-confidence constraints, trust is often a sufficient condition for making decisions. Applications often prove too difficult or too costly for formal analysis and inevitably fail to meet security requirements [3]. Knowing a remote party meets expectations provides sufficient assurance for continuing operation with them. Second, remote attestation frameworks are far simpler than applications they appraise and are more amenable to formal analysis. Ultimately, when developed using formal techniques we can precisely define what constitutes the trust they assess and that they do it correctly.

Working with colleagues at MITRE, JHUAPL and NSA we have collectively developed Copland [4] for representing, reasoning about, and executing remote attestation protocols. Copland specifies basic measurements for gathering evidence, mechanisms for sequencing measurements, and remote measurement requests for layering attestation. Copland also defines meta-evidence operators for evidence signing and hashing that make guarantees about that evidence. Signing is particularly critical to ensure evidence integrity and authenticity. Copland has a well-defined formal semantics for execution specifying the types of evidence produced and the precise order measurement activities are performed.

In prior work we defined Copland's formal semantics [4], defined and verified an execution environment for Copland that abides by the formal semantics [5], and illustrated usage patterns for layered attestation [6]. In this work we:

- Introduce a novel workflow for the design, execution, and formal analysis of Copland-based attestation protocols.
- Instantiate that workflow with a specific Copland protocol and accompanying security architecture to meet the

attestation needs for a UAV/Groundstation demonstration platform.

- Leverage automated analysis to characterize and constrain a capable adversary attempting to thwart that attestation.
- Exercise the Attestation Manager Monad environment to perform higher-level stateful actions like nonce-management, periodic attestation, and appraisal across executions of individual Copland protocols.

II. Copland

The primary aims of the Copland framework [4, 5, 7] are to support the specification, execution, and analysis of layered attestation protocols. Protocols are specified as language terms called *phrases*, and phrases serve as input to both execution and analysis alike. During execution phrases are interpreted to invoke local measurement routines, request remote measurements, and bundle evidence results cryptographically. During analysis phrases denote a precise measurement ordering and cryptographic evidence structure that, if upheld by an execution, constrains an adversary by the attacks it can perform and go undetected by attestation.

While complete descriptions of the Copland language and its semantics can be found elsewhere [4, 8] we describe here only what is relevant to understand the phrases presented in this work. The most primitive term in Copland concrete syntax is called a *Measurement Specification* and is a triple of the form: (S Q T) where S and T are symbols representing the measurement agent and target of measurement, respectively, and Q the place where T resides. As an example, the phrase:

```
(comp_hash Q dir)
```

is the composite hash, comp_hash, measurement performed over the contents of directory dir residing at place Q.

A simple extension to this protocol is:

```
(comp_hash Q dir) -> !
```

where the resulting hash evidence is sent to the signature operator, !, that signs it with a local place key. In Copland, a *place* is an abstract identifier for an attestation manager. All places have an associated key used for signing evidence that identifies the place. Putting it in context, the protocol where place P asks place Q to perform and sign a composite hash has the form:

```
P: @Q [(comp_hash Q dir) -> ! ]
```

The notation P: indicates P is the top-level place where this protocol is orchestrated and appraised. The @Q operator indicates a request to Q to execute measurements on P's behalf. Because the signature operation is performed in the context of Q, Q's key is used for signing. The result at P is a composite hash of dir at Q, both performed and signed by Q. This evidence can be appraised by checking the signature with Q's public key and comparing the hash with a known golden value.

Attestation protocols get far more interesting when layered among many principles. Consider a protocol from Helble et. al. [6] that performs a certificate-style attestation:

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

This protocol involves five places that gather, compose, and appraise evidence. P3 and P4 perform local attestations at the request of P1. Prior to making those requests, P1 performs a remote attestation of P3 and P4 to allow an appraiser to assess their fitness before trusting measurement they performed. P1 uses the linear sequence operator -> and parallel decomposition operator +~+ to ensure measurements of P3 and P4 occur before attestation requests, but allowing the requests to proceed in parallel. Furthermore, +<+ produces a sequential evidence product keeping attestation results separate.

It is not critical to understand the specifics of this layered background check protocol. However, the subtleties of attestation ordering, meta-evidence generation, and appraisal should be apparent. Ordering in particular is critical to appraisal and motivates our formal treatment and analysis of Copland phrases.

III. Copland Analysis Framework

In prior work we developed a formally-verified execution environment for Copland phrases called the Copland Virtual Machine (CVM) [5]. A crucial property of CVM execution is that, for a given Copland phrase, it emits a measurement event trace and cryptographic evidence which both respect the Copland reference semantics. In this work we additionally consider the security architecture of the platform where the CVM runs, and how formal properties of the CVM and its architecture bolster the soundness of a higher-level automated analysis.

Figure 1 shows our general framework for analyzing and executing an arbitrary Copland phrase t. The Appraiser platform takes t as input, sends an attestation request to the Target platform, then performs appraisal on the resulting evidence. The appraiser relies on a stateful execution environment provided by the Attestation Manager (AM) Monad (introduced in [5] and described further in Section V-D) to generate and remember a random nonce ensuring measurement freshness, construct an attestation request around that nonce, and finally perform appraisal.

The Target platform in Figure 1 leverages the CVM to carry out an attestation request by orchestrating system measurements and bundling evidence. During execution the CVM emits a trace of attestation-relevant events that correspond to t. A formal property proved about the CVM is that such a trace is guaranteed to respect a strict partial ordering on events called an *Event System*, derived statically from the phrase t. Precise ordering of measurement events is critical to constrain an active adversary [9].

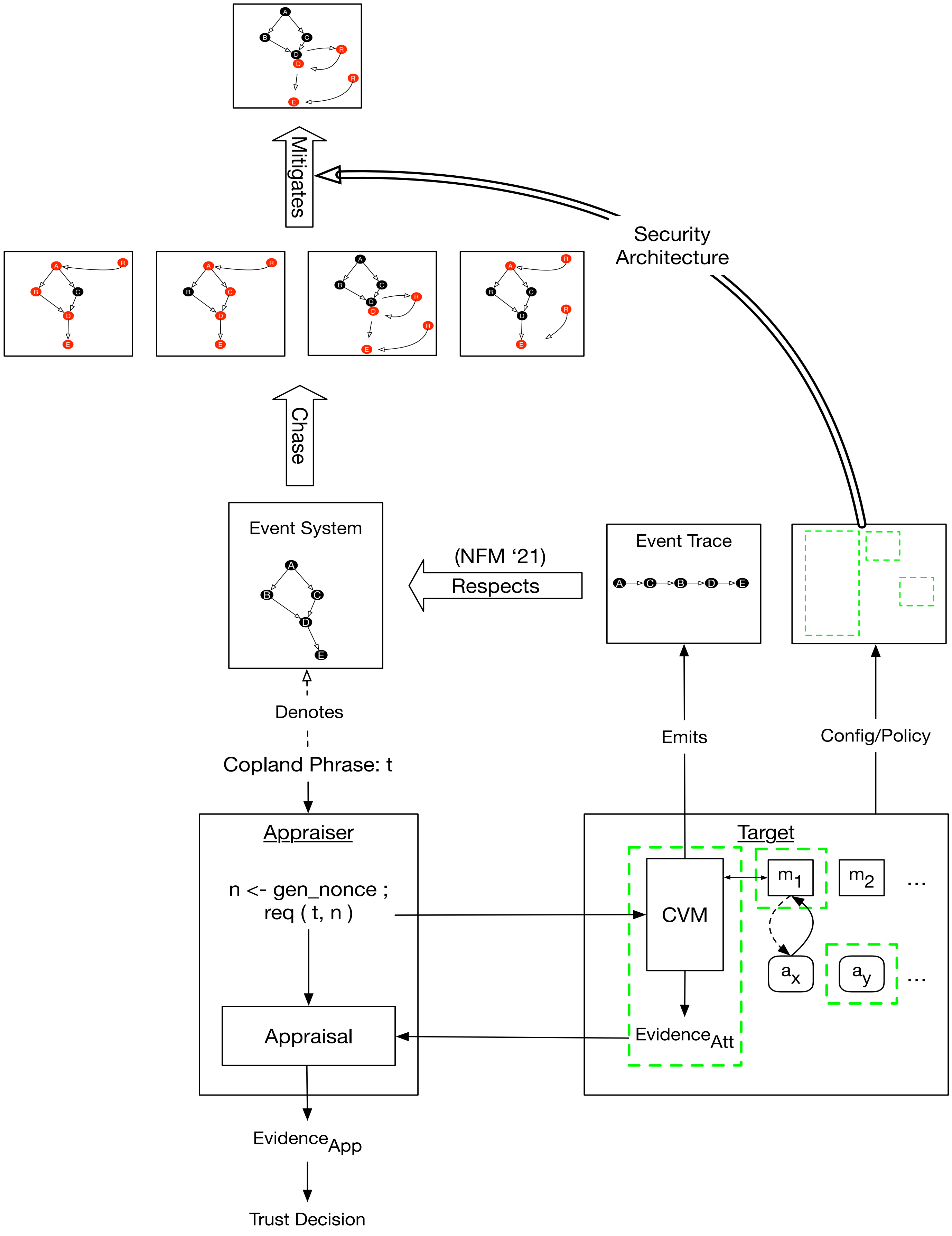

Fig. 1. Copland Verification Architecture

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 1 by dotted green lines around components. Such an architecture provides isolation and protects trusted measurers from their potentially-untrusted targets. In Section IV we instantiate a concrete measurement architecture based on the formally-verified seL4 microkernel [10]. However, our analysis framework remains parameterizable over the means of isolation for components at each layer of the target system.

Keeping properties of the Security Architecture abstract for analysis supports dropping in alternatives as deemed appropriate by the consumer of a particular attestation. Examples of alternative isolation mechanisms include trusted hypervisors, Intel's SGX enclaves [11], and SELinux [12], among others. Regardless of concrete insantiation, the existence of such a security architecture along with its policy and valid component implementations (e.g. the CVM and its measurement agents) should be apparent from the evidence returned by attestation. This goal aligns with the principle of *trustworthy mechanism* from Coker et. al. [1]. We discuss a concrete approach for achieving this via static measurement in later sections.

In addition to the execution of a Copland phrase t, Figure 1 shows an analysis flow for that same phrase. For analyis, the Event System derived from t serves to characterize honest measurement events in the context of a higher-level automated analysis. This analysis leverages a model-finding tool developed by our collaborators at MITRE instrumented explicitly to analyze Copland phrases [13]. Given a Copland phrase t, and an indication of the measurement target(s) of interest, the model finder will produce a set of models that describe all of the distinct ways an active adversary could corrupt/repair components during execution of t, corrupt the target(s) of interest, and go undetected by measurement.

One of the goals of a well-designed attestation system is that attacks like these should place a *high as possible* burden on the attacker. Towards this goal, our analysis framework incorporates properties of the security architecture to eliminate certain classes of attack from consideration. More concretely, architectural properties are encoded as logical assumptions to the model finder to indicate component corruptibility and the context (or lack thereof) that determines the integrity of components in relation to others. With these architectural assumptions incorporated, the attack models that remain must be acceptable to the consumer of attestation (i.e. sufficiently burdensome for an adversary). Otherwise, an iteration to refactor the Copland protocol and accompanying security architecture may be in order.

IV. Demonstration Platform

Our demonstration platform is an Un-piloted Air Vehicle (UAV) system taken from our work on the DARPA CASE program. We start with a pre-existing legacy implementation, extend it to support layered attestation built with Copland Attestation Managers, and formally analyze the resulting system. The transformed architecture exhibits desirable security properties that we can leverage as sound assumptions in the analysis framework from Figure 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 pre-existing UxAS [14] software, running in Linux. This software, if authentic, is considered trustworthy. Figure 2 shows the simple architectures of the ground station and UAV.

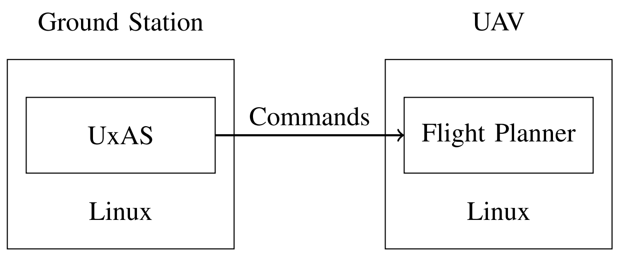

Fig. 2. Original UAV Architecture

In the initial model 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, while maintaining support of their legacy components.

A. Transformation of the UAV

In the new model, the UAV is augmented with a filter, paired with a Copland Attestation Manager (AM). The filter intercepts communication between the ground station and the UAV flight planner, forwards messages from trusted ground stations, and drops messages from un-trusted ground stations, as determined by the attestation manager. For a ground station that is neither trusted nor un-trusted (as is the case for unrecognized ground stations, or recognized ground stations whose trust-status has expired), the attestation manager will send it a Copland attestation request. If the ground station responds with corresponding evidence, then the attestation manager will appraise the evidence and provide a trustworthy/non-trustworthy decision to the filter. This extended UAV system is presented in Figure 3.

B. Transformation of the ground station

Where the UAV had to be extended to support appraisal, the ground station must be extended to support attestation, requiring more substantial architectural changes to the system. First, we add a Copland attestation manager to the Linux environment running UxAS. This attestation manager is outfitted with a number of Linux measurement procedures and UxAS-specific measurers to support attestation. The attestation manager is primarily responsible for monitoring the authenticity of the legacy ground station software and its outgoing messages.

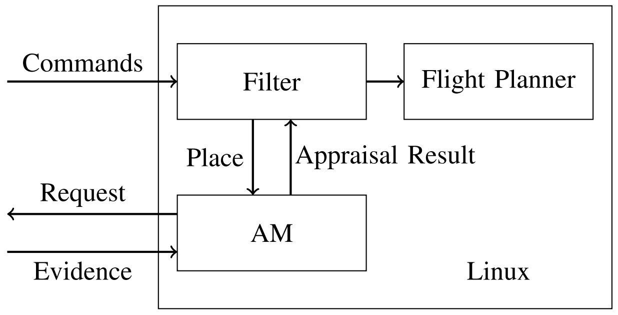

Fig. 3. Transformed UAV Architecture

Unfortunately, trust in this attestation manager is necessarily limited by its position in the Linux environment. It is running in the same Linux userspace as its target, the UxAS executable. It cannot be trusted to unearth deeply embedded threats such as a rootkit from the Linux userspace, because we cannot guarantee sufficient separation between the attestation manager and the equally privileged malicious actor. Such an actor could tamper with the attestation manager's measurements, or even steal its private key and impersonate it outright. For this reason, we virtualize the Linux environment, and add another layer to the system. In this new model, the ground station runs the seL4 microkernel, with two components. One is a virtual machine manager, hosting the aforementioned Linux environment. The other component is an additional attestation manager. To distinguish the two, we refer to the attestation manager running in the Linux environment as the *UserAM*, and the attestation manager running at the seL4 level as the *PlatformAM*.

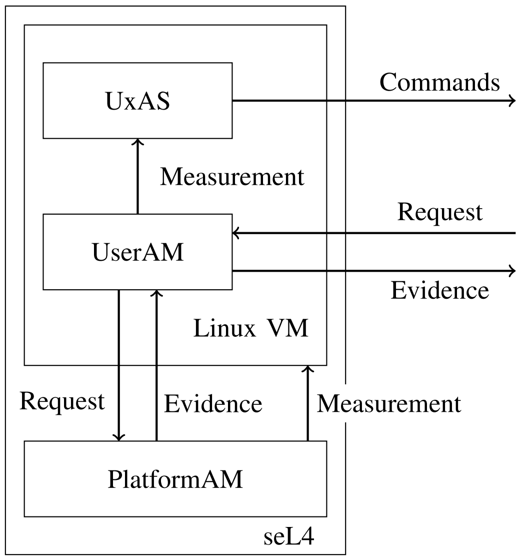

Fig. 4. Transformed Ground Station Architecture

Introduction of the seL4 layer allows the PlatformAM to externally conduct measurements of the vulnerable Linux environment. Crucially, seL4 possesses strong memory separation capabilities which prevents a compromised and malicious Linux kernel from interfering with the PlatformAM's measurements. These memory access controls are part of the specification of seL4 that has been formally verified, allowing for provably-separate components [15].

The two attestation managers work together in a layered attestation paradigm. The UserAM can make granular measurements easily from within the Linux environment, while the PlatformAM can attest to the healthiness of the Linux VM as a whole. In this sense, the PlatformAM extends its increased trustworthiness to the UserAM after measuring its Linux environment.

Finally, we must consider what supports the trustworthiness of the PlatformAM and the seL4 layer in general. The memory separation properties of the seL4 layer are static guarantees, so we can trust in this layer so long as the correct seL4 image was booted. To accomplish this, we need to anchor our trust of the PlatformAM in some hardware-based root-of-trust. This evidence will necessarily be hardware-specific. For our design purposes, we assume that the boot process leaves some evidence token available to the PlatformAM which convincingly indicates that the proper seL4 image was booted.

C. Key Release

Each attestation manager possesses a private key it uses to endorse evidence collected during Copland protocol execution. The semantics of Copland makes the simplifying assumption that each attestation manager has exclusive access to their private key to strongly identify evidence bundles they generate. However, in our concrete design the attestation managers do not begin execution with posession of said keys. Instead, keys are strategically released during the start-up process to avoid starting a compromised attestation manager and thereby leaking its key. Assurance that only good attestation managers possess their keys together with the separation guarantees of seL4 provide suitable evidence in support of the private key assumption.

Key release is performed in a "bottom-up" manner. For this system, the PlatformAM is the lowest-layered attestation manager, and so its key will be released first. It begins execution with an encrypted key, which it then decrypts with the root-of-trust token. Note that if an improper seL4 image was booted, potentially containing a comprompsed PlatformAM, then the root-of-trust token would be incorrect. Therefore, any attempt of the PlatformAM to acquire its key would fail.

Next, the PlatformAM prepares to release the UserAM's key. Before it does so, it must first measure the UserAM and the encompassing Linux environment to be satisfied of their authenticity. These measurements may be the same Copland protocols an appraiser might request from the attestation manager at runtime. Once the PlatformAM is confident that the UserAM is trustworthy, it releases its key by providing it with a decryption key. This decryption key is derived from the PlatformAM's private key, in order to ensure that a compromised PlatformAM cannot release the UserAM's key.

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

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

V. Transformed UAV Platform Analysis

Using the Copland Verification Architecture we will analyze the UAV attestation architecture. We will first describe the Copland phrase and its components, then derive properties of honest measurement from the CVM semantics for that phrase. Next we encode security properties of the UAV architecture as custom logical assumptions passed as additional input to the model finder analysis for the phrase. Finally, we will analyze attack models generated for the phrase as a whole, then consider the tradeoffs between alternative measurement strategies that differ in depth and frequency.

A. *Copland Phrase Description/Components*

A Copland phrase to measure the transformed ground station target platform appears in Figure 5. The phrase begins with: $*\mathsf{heliAM}, \mathsf{n}:$ which designates heliAM (the UAV AM component from Figure 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. Recall from Figure 4 that UserAM and PlatformAM are distinct attestation domains both running on the remote ground station platform. We represent these in Copland as place identifiers userAM and platAM.

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

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

Fig. 5. UAV Copland Phase

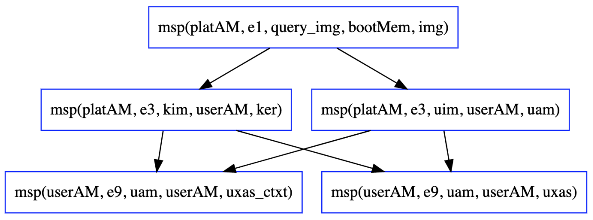
msp(platAM, e1, query_img, bootMem, img)
msp(platAM, e3, kim, userAM, ker)
msp(platAM, e3, uim, userAM, uam)
msp(userAM, e9, uam, userAM, uxas_ctxt)
msp(userAM, e9, uam, userAM, uxas)

Fig. 6. Event System derived from the Copland phrase in Figure 5

(uim userAM uam) that specifies an integrity measurement of the uam (Userspace Attestation Manager) component that will in turn perform more specialized measurements of the UxAS application. 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 more specialized measurements of the target application. Descriptions of all measurements and their targets can be found in Table I. 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 (via the ! operator) and sends it in a response back to heliAM.

B. *Event Semantics*

Figure 6 shows the Event System, a partial ordering on measurement events, determined by the Copland phrase in Figure 5. 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 which is formally verified to uphold such measurement orderings.

C. *Architectural Assumptions*

Given the measurement events in Figure 6, the model finding tool requires additional assumptions about the environment

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

Fig. 7. Assume adversary avoids detection at main measurement event.

in which these measurements are carried out before it can produce a meaningful analysis. The first of these assumptions 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 the target while avoiding detection. Figure 7 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.

Given just honest measurement events and the 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 8. 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 of specific components and properties of the architecture where they operate. The first two statements in Figure 9 encode that components at bootMem and platAM do not depend on any other components 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. The first statement of Figure 10 says that only way to corrupt a component at platAM is by corrupting img at bootMem. 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 that is highly scrutinized for absence of vulnerabilities. Adding these assumptions brings the number of attack models down to 35.

As we add assumptions about the integrity of individual components and their environment, the attack space continues to shrink. After adding the last two statements in Figure 10 about component integrity, the number of distinct attacks goes down to 14. One says that the boot image cannot become corrupted. This is justified by protecting the image somewhere like trusted hardware or a dedicated seL4 component, where only highly-priviledged bootloader code has write access. The second says that the only way for uam to become corrupted is via a corrupted OS kernel. To justify this in our prototype, we limit the code of uam to very specific measurement functions,

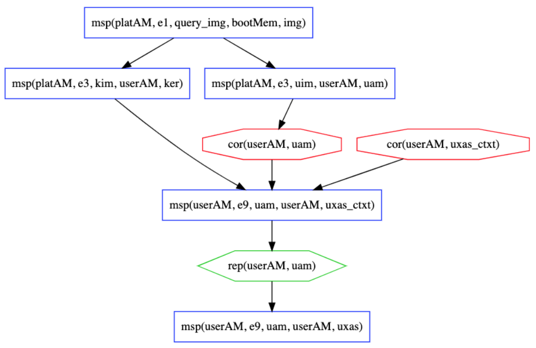
msp(platAM, e1, query_img, bootMem, img)
msp(platAM, e3, kim, userAM, ker)
msp(platAM, e3, uim, userAM, uam)
cor(userAM, uam)
cor(userAM, uxas_ctxt)
msp(userAM, e9, uam, userAM, uxas_ctxt)
rep(userAM, uam)
msp(userAM, e9, uam, userAM, uxas)

Fig. 8. Corruption and Repair of uam

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

Fig. 9. Contextual assumptions about "deep" components in the architecture.

and include them as a library packaged with the Copland Virtual Machine at userAM. This assumption effectively deems dynamic attacks on uam out of scope unless they rely on a corrupted (or mis-priviledged) OS kernel to succeed. One such attack would be replacing the uam or CVM executables with rogue versions at runtime.

Figure 11 defines four final assumptions that further shrink the 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 such as linux are feasible in practice, one could toggle this assumption off to explore the implications of such an attack. However, for our final analysis we assume that the (kim userAM ker) measurement lauched 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 models in Figure 12 characterize the two remaining ways to corrupt uxas and go undetected. Specifically, the adversary must corrupt uxas or its context *after* they are measured and before the flight planning services of uxas are consumed. This is an example of a recent or time-sensitive attack that is in theory more difficult to perform[9]. We next discuss strategies for making these types of attacks more difficult still by repeated measurement,

```
% platAM components only corrupted via a corrupt boot image (img)
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).
```

Fig. 10. Assumptions about the "corruptibility" of components.

```
% 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.
```

Fig. 11. Final Architecture Assumptions

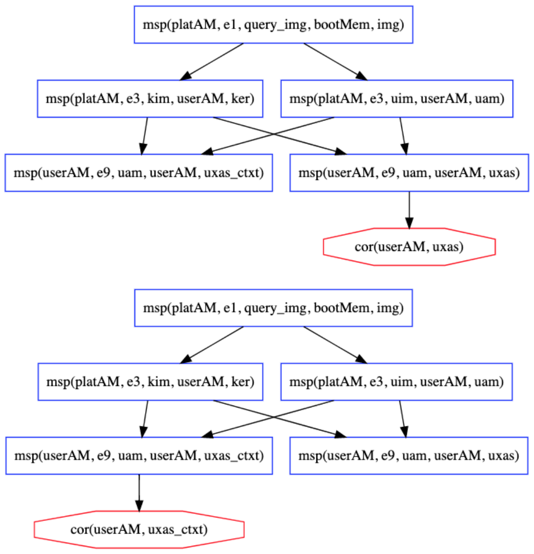
msp(platAM, e1, query_img, bootMem, img)
msp(platAM, e3, kim, userAM, ker)
msp(platAM, e3, uim, userAM, uam)
msp(userAM, e9, uam, userAM, uxas_ctxt)
msp(userAM, e9, uam, userAM, uxas)
cor(userAM, uxas)
msp(platAM, e1, query_img, bootMem, img)
msp(platAM, e3, kim, userAM, ker)
msp(platAM, e3, uim, userAM, uam)
msp(userAM, e9, uam, userAM, uxas_ctxt)
msp(userAM, e9, uam, userAM, uxas)
cor(userAM, uxas_ctxt)

Fig. 12. Final Attack Models

and thus justify leaving them as acceptable attack models.

D. AM Monad alternatives

An individual Copland phrase, when executed inside the Copland Virtual Machine, emits a precise sequence of measurement events and results in an evidence bundle of predictable shape. From the appraiser's perspective the protocol is executed in one shot, meaning the entire phrase is executed depth-first, by all places involved, before evidence bubbles back to the appraiser. 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, perhaps smaller, Copland phrases and compose intermediate evidence results.

Put another way, pure Copland has no state that persists across executions of individual protocols. While generally useful, state is critical in the context of end-to-end attestation to preserve nonce values, evaluate evidence for trustworthiness, and take actions according to that evidence. Similarly, Copland has no error handling mechanism to account for failed or divergent attestations. To remedy these we have defined the *Attestation Manager (AM) Monad* as a standard state monad with exceptions, with a formal definition in Coq [16] and prototype implementations in Haskell and CakeML[17, 18].

An example computation in the AM Monad called `attest_gs` that prepares, executes, and appraises a Copland phrase from the UAV scenario is 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 used to measure the groundstation target. Assuming the Copland phrase from earlier in Figure 5 is assigned the name `case_cop`, we could instantiate this AM Monad computation via function application: `attest_gs case_cop`.

Recall from earlier that even with 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 some time interval chosen by the appraiser to make attack and repair of the target more difficult for the adversary. Depending on the scenario and capabilities of the attacker, `s` could be restricted under a certain threshold, or even randomly generated within some range.

However, a problem arises if the time required to complete all measurements of the phrase exceeds `s`. Recall that this phrase involves measurements of deep components that may require more time to complete. Especially in real-time embedded systems with hard scheduling requirements, attestation services might be given constrained resources including CPU time to complete measurement tasks [19] .

Deeper measurements tend to stall or freeze the system to capture its state. To accomodate such requirements, it may be that performing one deep measurement of the system upfront is enough to establish a baseline, and repeated shallow measurements are then sufficient to maintain evidence of integrity. Towards this goal, a partitioning of the phrase into its deep and shallow portions is as follows:

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

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

Measurer	Description
query_img	Queries the contents of the static boot image (has read-only permissions).
kim–Kernel Integrity Measurer	Measures dynamic snapshots of OS kernel data structures.
uim–Userspace Integrity Measurer	Measures generic properties of targets in userspace memory. An example is a measurement of \proc to check for dynamic substitution of a rogue binary.
uam–Userspace Attestation Manager	A library of specialized measurement procedures of components in userspace.

Target	Description
img	Contents of the static boot image.
ker	OS kernel data structures.
uxas_ctxt	The execution context of UxAS. This includes things like dynamic configuration files and the network stack used for communication.
uxas	The application of interest. Legacy flight planning software.

Place	Description
bootMem–Boot Memory	Protected storage location holding a copy of the seL4 image loaded onto the platform at boot time. Boot loader has exclusive write-access, components at platAM have read-only access.
heliAM–Helicopter Attestation Manager	An Unmanned Air System platform attempting to establish trust in a remote ground station before accepting flight commands from that ground station.
platAM–Platform Attestation Manager	Running instance of the Copland Virtual Machine in a privileged domain on the ground station platform. Runs as a native seL4 CamKEs component, and has read access to memory at the bootMem and userAM places.
userAM–Userspace Attestation Manager	Running instance of the Copland Virtual Machine at the top-level of the ground station platform. Runs in a virtualized linux environment alongside the uxas target.

TABLE I
PHRASE COMPONENTS

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)
  )
)
```

Implicit in these definitions are potentially-failing attestations. A discerning appraiser could implement exception handlers to interpret the different ways attestation can fail. For timeouts perhaps resending the request is sufficient, but evidence indicating platform compromise may warrant notifying the UAV filter component to restrict subsequent communication with that groundstation.

In addition to the formal definition of the AM Monad in Coq, formal verification of facts about nonce handling and appraisal are ongoing. Prototype implementations exist in Haskell [17] and CakeML [18]. Because CakeML does not provide built-in monadic libraries, we mimic AM Monad functionality with mutable state. In future work we hope to close the gap between the implementation and formal definition by synthesizing such stateful code from its monadic counterpart in Coq, borrowing the approach outlined in [20].

VI. RELATED WORK

Although many tools and frameworks for remote attestation and related trusted computing technologies have emerged over the years, comparitavely few involve formal analysis of attestation protocols and their architecture. Nunes et. al. [21] developed VRASED (Verifiable Remote Attestation for Simple Embedded Devices) as the first formal verification "of a HW/SW co-design implementation of any security service". They use LTL to verify end-to-end security and attestation soundness by composing properties of Verilog hardware specifications with properties derived from cryptographic software. The author's more recent APEX framework [22] extends the security architecture of VRASED to support unforgeable remote proofs of execution (PoX) on low-end devices. They achieve convincing end-to-end security guarantees for a fixed embedded platform. In contrast, our Copland-based analysis supports more diverse, layered attestation scenarios and experimenting with different protocol/architecture designs with respect to a powerful adversary.

Sardar et. al. formally specify and verify properties of specialized attestation primitives used in Intel's SGX and TDX technologies [23, 24, 25]. They encode the primitives as models in the ProVerif tool and perform symbolic analysis over the protocols with respect to a Dolev-Yao adversary.

Rowe's work on analysis of layered attestation protocols characterizes formally how measurement and evidence bundling strategies force an adversary to perform more difficult deep or recent attacks [9, 26]. Besides influencing the design of our UAV attestation protocol and architecture, the formalizations in this work led to proofs of soundness for both honest and adversarial axiom extensions to the Chase model finder that we use for analysis [13].

The Copland language definition and reference semantics [4] serves as the formal foundation for both analysis and execution of layered attestation protocols in this work, and the Copland Virtual Machine [5] provides an execution environment for Copland phrases verified in Coq with respect to the reference semantics. Helble et. al. [6] demonstrate the complexity of the design space for Copland-based attestation protocols and motivate our frameowrk for analysis. On the implementation side, Maat [27] is of note as an attestation framework that motivated the design of the Copland language and the Copland virtual machine.

The model finder tool was recently released publicly as part of the Copland Collection [8] and is described in depth in a publication currently in press [13]. The tool instruments the more general Chase model finder [28] with specialized axioms that encode the semantics of Copland measurement and the behavior of active adversaries that can corrupt and repair components relevant to Copland attestations.

VII. Future Work/Conclusion

In this work we describe a prototypical system-level design process that integrates formal methods at critical decision points. The system-level goal is integrating a remote attestation system into a legacy system to establish trustworthiness. The formally verified Copland language and interpreter are used to define an attestation protocol. The Attestation Manager Monad situates the interpreter in an execution environment providing nonce management and appraisal. With a verified environment in place, MITRE's model finder tool is used to explore the allowed behaviors of an adversary to help in refining the attestatin protocol. Finally, the attestation manager monad is used explore various decompositions of the attestation protocol. The result of this work is not an ideal proof-of-correctness in a single formal tool, but a collection of evidence from multiple tools integrated into a design process that can be revisited as the UAV system changes or applied to alternate designs.

The future of Copland and our design tools includes synthesis of implementations from Coq, integration of hardware-based static measurements, and generalizing our approach to other attestation architectures. The CakeML implementation of the attestation manager will be synthesized from its Coq specification to further ensure correctness. Architectural properties like seL4 component separation and key management could be captured more formally and automatically integrated into our analysis pipeline rather than our manual encoding of assumptions for the UAV scenario. Finally, we will begin building an attestation testbed that will allow empirically evaluating multi-agent attestation frameworks.

References

[1] 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*, vol. 10, no. 2, pp. 63–81, June 2011.

[2] V. Haldar, D. Chandra, and M. Franz, "Semantic remote attestation – a virtual machine directed approach to trusted computing," in *Proceedings of the Third Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

[3] 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*, 1998, pp. 303–314.

[4] 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.

[5] A. Petz and P. Alexander, "An infrastructure for faithful execution of remote attestation protocols," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez, Eds., vol. 12673. Berlin, Heidelberg: Springer International Publishing, 2021, pp. 268–286.

[6] S. Helble, I. Kretz, P. Loscocco, J. Ramsdell, P. Rowe, and P. Alexander, "Flexible mechanisms for remote attestation," *ACM Transactions on Privacy and Security*, to appear.

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

[8] C. Authors, "Copland website," https://ku-sldg.github.io/copland, 2021.

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

[10] 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*, vol. 53, no. 6, pp. 107–115, 2010.

[11] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2487726.2488368

[12] F. Mayer, K. MacMillan, and D. Caplan, *SELinux by Example*. Prentice Hall, 2007.

[13] P. Rowe, J. Ramsdell, and I. Kretz, "Automated trust analysis of copland specifications for layered attestations," in *To Appear In: Principles and Practice of Declarative Programming (PPDP 21)*, Sep. 2021.

[14] S. Balachandran, C. A. Muoz, M. C. Consiglio, M. A. Feli, 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)*, 2018, pp. 1–6.

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

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

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

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

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

[20] 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 *Automated Reasoning - 9th International Joint Conference (IJCAR)*, ser. Lecture Notes in Computer Science, D. Galmiche, S. Schulz, and R. Sebastiani, Eds., vol. 10900. Springer, 2018, pp. 646–662. [Online]. Available: https://cakeml.org/ijcar18.pdf

[21] 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*, ser. SEC'19. USA: USENIX Association, 2019, pp. 1429–1446.

[22] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 771–788. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/nunes

[23] M. U. Sardar, R. Faqeh, and C. Fetzer, "Formal foundations for intel SGX data center attestation primitives," in *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, ser. Lecture Notes in Computer Science, S. Lin, Z. Hou, and B. P. Mahony, Eds., vol. 12531. Springer, 2020, pp. 268–283. [Online]. Available: https://doi.org/10.1007/978-3-030-63406-3_16

[24] M. U. Sardar, D. L. Quoc, and C. Fetzer, "Towards formalization of enhanced privacy ID (epid)-based remote attestation in intel SGX," in *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*. IEEE, 2020, pp. 604–607. [Online]. Available: https://doi.org/10.1109/DSD51259.2020.00099

[25] M. U. Sardar, S. Musaev, and C. Fetzer, "Demystifying attestation in intel trust domain extensions via formal verification," *IEEE Access*, vol. 9, pp. 83 067–83 079, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3087421

[26] P. D. Rowe, "Bundling Evidence for Layered Attestation," in *Trust and Trustworthy Computing*. Cham: Springer International Publishing, Aug. 2016, pp. 119–139.

[27] J. A. Pendergrass, S. Helble, J. Clemens, and P. Loscocco, "A platform service for remote integrity measurement and attestation," in *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, 2018, pp. 1–6.

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