A Copland Attestation Manager

Adam Petz, Perry Alexander

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

ABSTRACT

Copland is a domain specific language designed for describing, analyzing and executing attestation protocols. Its formal semantics defines evaluation, sequencing, and dispatch of measurements resulting in evidence describing a system's state. That evidence is in turn appraised to determine if and how an external system will interact with it. The contribution of this work is a description of the first Copland interpreter and the attestation manager built around it. Following an overview of the syntax and formal semantics is a collection of motivating examples. Next is a description of a Haskell-based Copland interpreter and the attestation manager constructed around it. Examples are provided to show the interpreter's interface format. A description of the Copland landscape and future goals closes the presentation.

CCS CONCEPTS

• **Security and privacy** → *Authentication*;

KEYWORDS

remote attestation, attestation protocol manager

1 INTRODUCTION

Remote attestation is the activity of making a claim about properties of a target by supplying evidence to an appraiser over a network [3]. In its simplest form, a remote attestation process consists of an *appraiser* that sends an *attestation request* to a *target* that performs an attestation and responds with *evidence* describing desired properties (Figure 1). The appraiser evaluates evidence to determine if the target is trustworthy while the target protects its secrets in the spirit of a zero-knowledge proof system [5].

Evidence returned by an attestation consists of descriptive evidence and meta-evidence. Descriptive evidence traditionally includes hashes, TPM PCRs [20], and PCR composites, constructed
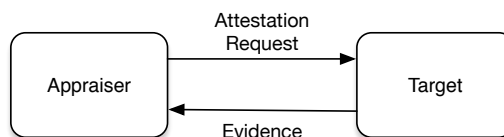
Figure 1: Basic attestation architecture.

by measuring target components. Meta-evidence includes nonces, signatures, and certificates to ensure freshness, integrity, and authenticity of evidence. While evidence describes the target system, meta-evidence certifies the evidence gathering process.

A simple attestation example useful as a running example is determining that a target system is running a virus checker. A naive protocol returns a signed evidence package that includes a nonce for freshness, N_0, and the hash of the virus checking binary, vc. The interaction between appraiser A and target B is expressed using a canonical protocol notation as:

$$A \rightarrow B : N_0$$
$$B \rightarrow A : [\![\#vc, N_0]\!]_B$$

A sends N_0 to B initiating the attestation. B responds with the hash of vc, #vc, and N_0 together signed using its private key. A then checks the evidence signature, checks the nonce, and compares the virus checking software hash with a known good value. This simple appraisal guarantees freshness, integrity and authenticity, and good measurement. Importantly, A gains no knowledge of B other than it produced the correct hash value.

Attestation becomes more complicated with the introduction of layered attestation and evidence bundling [16], mutual attestation, non-trivial contextual measurement [11], and stateful measurement [1, 20]. As a result attestation is frequently implemented using protocols that perform measurement, gather evidence, and prepare meta-evidence. An appraiser and target must agree on a protocol to execute before initiating an attestation.

One such protocol performs a more detailed appraisal of the virus checker. The appraiser will again send a nonce and request an attestation. The appraiser in this case wants information about virus checker's signature file and signature file server as well as executable code. As an added challange, the appraiser may not have details of the signature file server and must rely on the target to perform appraisal. A protocol for such a transaction might be represented as follows:

Adam Petz, Perry Alexander

$$A \;\leftarrow\; \text{USM } \bar{a} \mid \text{KIM } P \, \bar{a} \mid \text{CPY} \mid \text{SIG} \mid \text{HSH} \mid \cdots$$
$$T \;\leftarrow\; A \mid @_P T \mid (T \to T) \mid (T \stackrel{\pi}{<} T) \mid (T \stackrel{\pi}{\sim} T)$$
$$E \;\leftarrow\; \xi \mid \text{U}_P(E) \mid \text{K}_P^P(E) \mid [\![E]\!]_P \mid \#_P E \mid$$
$$\quad (E \;;; E) \mid (E \parallel E) \mid \cdots$$

where $\pi = (\pi_1, \pi_2)$ is a pair of splitting functions.

Figure 2: Term syntax.

$$A \to B : N_0$$
$$B \to S : N_1$$
$$S \to B : [\![\#s, N_1]\!]_S$$
$$B \to A : [\![\#vc, \#sf, \text{app}([\![\#s, N_1]\!]_S), N_0]\!]_B$$

In this *attestation protocol* A remains the appraiser while B is the target. A sends a nonce to B to initiate attestation and B immediately sends a new nonce to S, the signature file server, to initiate its attestation protocol. S returns a signed hash with the nonce from B. B returns a signed hash of its virus checker binary, its signature file, and the result of B's appraisal of evidence from S. A appraises the evidence by checking B's signature, checking the original nonce, checking hashes, and checking the appraisal result.

There are a number of potential problems and design decisions that arise when designing even simple protocols like these examples. Nonce reuse, caching server evidence, server appraisal by the target or appraiser, and ordering measurements and appraisals all play a role in design and verification. Ad hoc mechanisms are not suitable for capturing nuances of even simple attestation protocols. Providing mechanisms for representing, verifying, and executing attestation protocols is the focus of the Copland effort.

2 COPLAND

Copland is a domain specific language and formal semantics for describing attestation protocols. It is designed to address specific issues of ordering and remote execution critical to attestation protocol design. Copland's semantics are defined as a deep embedding in Coq [2]. Terms (called *phrases*) define individual protocols. Evidence defines the *form* of evidence produced by protocol execution. A denotational semantics maps terms to the evidence they produce while an event-based operational semantics defines system events associated with protocol execution. Together they define both correct evaluation results and proper event ordering during evaluation. A brief description of Copland's formal semantics is necessary to understand the attestation manager written around it.

2.1 Terms and Evidence

Formal syntax definitions for Copland terms and evidence appear in the grammar of Figure 2 while a denotational semantics relating terms to evidence is defined in Figure 3. Informally there are four term types in Copland: (i) measurements; (ii) operations; (iii) execution sequencing; and (iv) requests.

The meta-variable P appears throughout the term grammar and represents a *place*. In Copland places represent any location where attestation protocols are interpreted. Each place has a unique private key and a policy identifying how it performs measurements. In Copland implementations, P is always associated with an attestation protocol interpreter as described later.

The terms (USM $\bar{a}$) and (KIM P $\bar{a}$) represent measurements and are held abstract in the language. (USM $\bar{a}$) is a *User Space Measurement* that performs a measurement in the local user space. $\bar{a}$ abstractly represents the details of the measurement operation. Such measurements include hashing a file, requesting a TPM quote, or examining the local /proc directory contents. For example, the term (USM *hash* /*etc*/*passwd*) might be used to request a hash of a system's password file. In contrast, (KIM P $\bar{a}$), is a *Kernel Integrity Measurement* that examines some external system. An example of such a measurement is running LKIM [11] on a virtual machine's Linux kernel. $\bar{a}$ abstractly represents the the measurement operation while P represents the place measured. The term (KIM 5 *LKIM*) might be used to request that LKIM be run on the kernel associated with place 5.

The terms CPY, SIG, and HSH are a minimal set of evidence operations. Measurements produce evidence while these operations process evidence by copying, signing, and hashing evidence values, respectively. Copland's design supports adding other operations similarly, as long as they consume and produce evidence. An alternative to adding language primitives is to encode new operations as custom USM procedures. This approach has the advantage of keeping the language smaller, but may harm protocol readability and complicate static analysis.

The terms $(T_1 \to T_2)$, $(T_1 \stackrel{\pi}{<} T_2)$, and $(T_1 \stackrel{\pi}{\sim} T_2)$ compose terms and impose ordering on execution. $(T_1 \to T_2)$ composes two terms and specifies that T_2 strictly follows T_1 and consumes evidence it produces. $(T_1 \stackrel{\pi}{<} T_2)$ also specifies sequential execution, but it splits evidence between the terms using the pair of projection functions π. As a common example, $\pi = (id, \perp)$ would route all prior evidence to T_1 and empty evidence to T_2. $(T_1 \stackrel{\pi}{\sim} T_2)$ also splits evidence, but allows the terms to execute in parallel.

Finally, $@_P(T)$ is a communication primitive requesting that a term be evaluated at another place. Similar to a remote procedure call, $@_P(T)$ requests that a place interpret a specified protocol and return the resulting evidence. The executed protocol will frequently be subject to negotiation using policy as context.

Evidence produced by protocol execution includes basic values and composition operators that indicate order and place. ξ represents empty evidence. By definition no Copland term can produce ξ, but it is useful as initial evidence. $\text{U}_P(E)$ and $\text{K}_P^P(E)$ are evidence values produced by USM and KIM, respectively. E is for evidence accumulated prior to executing the USM or KIM, and both record the place targeted by the associated measurement. KIM separately records the place performing the measurement while for USM the same place plays both roles. The proper way to read $\text{K}_Q^P(E)$ is "the result of a KIM measurement of place P performed by place Q". It is worth noting that the evidence semantics for USM and KIM terms does not retain the list of arguments, $\bar{a}$. This does not cause problems for analysis since the appraiser can always recover $\bar{a}$ from the originating Copland term. We *do* include the argument lists in the concrete evidence datatype in our Haskell implementation, but this is simply an implementation quirk for convenience. $[\![E]\!]_P$

$$
\begin{aligned}
\mathcal{E}(\text{USM } \bar{a}, p, e) &= \mathsf{U}_p(e) \\
\mathcal{E}(\text{KIM } q\ \bar{a}, p, e) &= \mathsf{K}_p^q(e) \\
\mathcal{E}(\text{CPY}, p, e) &= e \\
\mathcal{E}(\text{SIG}, p, e) &= [\![e]\!]_p \\
\mathcal{E}(\text{HSH}, p, e) &= \#_p\, e \\
\mathcal{E}(@_q\, t, p, e) &= \mathcal{E}(t, q, e) \\
\mathcal{E}(t_1 \to t_2, p, e) &= \mathcal{E}(t_2, p, \mathcal{E}(t_1, p, e)) \\
\mathcal{E}(t_1 \overset{\pi}{\prec} t_2, p, e) &= \mathcal{E}(t_1, p, \pi_1(e)) \mathrel{;;} \mathcal{E}(t_2, p, \pi_2(e)) \\
\mathcal{E}(t_1 \overset{\pi}{\sim} t_2, p, e) &= \mathcal{E}(t_1, p, \pi_1(e)) \parallel \mathcal{E}(t_2, p, \pi_2(e)) \\
&\quad \text{where } \pi = (\pi_1, \pi_2)
\end{aligned}
$$

Figure 3: Evidence Semantics.

and $\#_P E$ are the results of signing and hashing, again recording the place performing the operation. Finally, $(E \mathrel{;;} E)$ and $(E \parallel E)$ compose evidence taken in sequence and parallel, respectively.

2.2 Denotational Semantics

Figure 3 defines a denotational semantics, $\mathcal{E}(t, p, e)$, mapping each Copland term, t, initial evidence value, e, in some place, p, to a resulting evidence term. In the formal Copland specification this is called the *evidence semantics*, and it provides a requirements definition for Copland interpreters. This semantics captures the structure of the gathered evidence rather than concrete mesurement results. In this sense, it might be more accurate to call $\mathcal{E}$ an evidence *type assignment*. A major outcome of our Haskell implementation (described later) was the need to add concrete measurement values to the evidence datatype. To what extent the properties of these values make their way back to the formal semantics remains an open research question. For the examples in the following section we use the concrete evidence representation rather than the formal evidence semantics in order to demonstrate concrete measurement results. For example, $@_1 (\text{USM } hash \text{ vc})$ results in $\mathsf{U}_1(\#vc)$ rather than $\mathsf{U}_1(\xi)$ as it would in the formal semantics.

2.3 Examples

It is possible to encode various attestation processes associated with our virus checker example in Copland terms. The simplest action is to ask place 1 to return a hash of its virus checker:

$$@_1 (\text{USM } hash \text{ vc})$$

Adding meta-evidence to help assure authenticity, ask place 1 to return a signed hash of vc:

$$@_1 ((\text{USM } hash \text{ vc}) \to \text{SIG})$$

A layered attestation asks place 1 to determine the integrity of place 2 and then asks place 2 to return a hash of its signature server, ss:

$$@_1 ((\text{KIM } 2\ LKIM) \overset{\pi}{\prec} @_2 (\text{USM } hash \text{ ss}))$$

This protocol implements an important pattern used to chain trust. 1 performs an LKIM measurement on 2 to gather evidence and then

asks 2 to perform a measurement. During appraisal 1 can assume that 2's measurement is trustworthy if the LKIM measurement shows that 2 itself is trustworthy. Continuing this process, 2 may then do the same to 3. If the chain of LKIM measurements is sound, then 3 is trustworthy because its behavior was observed by a trustworthy 2 whose behavior is observed by a trustworthy 1, assuming 1 is a root-of-trust.

One might consider making the previous request more efficient by performing the LKIM measurement and the signature server measurement in parallel:

$$@_1 ((\text{KIM } 2\ LKIM) \overset{\pi}{\sim} @_2 (\text{USM } hash \text{ ss}))$$

Unfortunately, the parallel protocol is semantically quite different than the sequential protocol. The sequential protocol assures that 1's LKIM measurement of 2's kernel occurs strictly before 2's measurement of its signature server. This ordering of measurements gives place 1 high confidence that 2's measurement was performed on the system associated with the preceding LKIM measurement. An adversary could theoretically corrupt the kernel at place 2 between the two measurements to avoid detection. However, this ordering puts more burden on the adversary to corrupt a deeper, more protected kernel component and perform the attack in a small time-of-check-time-of-use window [16, 17]. If the measurements are performed in parallel, this inference cannot be made. If the LKIM measurement occurs after the signature server measurement we know nothing of 2's state when the signature server is hashed. Regardless of what 1 learns from the LKIM measurement, the chain of trust from 1 to 2 is broken.

To ensure freshness of the virus checker measurement, USM N_0 introduces a nonce that becomes the initial evidence passed to the $@_1$ term:

$$@_0 (\text{USM } \mathsf{N}_0) \to @_1 ((\text{CPY } \overset{\pi}{\prec} (\text{USM } hash \text{ vc})) \to \text{SIG})$$

where $\pi = (id, \bot)$ splits the nonce over the two branches of the $\prec$ compsition. id sends the nonce to CPY while $\bot$ sends nothing to the USM. The resulting evidence is $\mathsf{U}_0(N_0) \mathrel{;;} \mathsf{U}_1(\#vc)$. The last term in the sequence causes place 1 to sign the result generating the evidence:

$$[\![\mathsf{U}_0(N_0) \mathrel{;;} \mathsf{U}_1(\#vc)]\!]_1$$

Place 0 can appraise the result by checking place 1's signature on the evidence, then checking the nonce and the hash value. Again this is an important attestation protocol pattern where signed evidence that includes a nonce provides evidence of freshness, integrity, and authenticity.

Mutual attestation occurs when both places request attestations and appraise results. An initial example has place 1 executing a term from place 0 that asks place 0 for an attestation while performing its own attestation:

$$@_0 (@_1 (@_0 (\text{USM } hash \text{ am}) \sim (\text{USM } hash \text{ am})))$$

In agreeing to execute this protocol, place 0 is authorizing place 1 to request an attestation while responding to its original request. In this case, both place 0 and place 1 hash their respective attestation managers. When the protocol completes, each place will have a hash of the other. The attestation protocol is mutual because both places are playing the role of appraiser and target.

This protocol has place 1 "in charge" of the attestation process by initiating measurement of place 0. In contrast, place 0 initiates attestation in the next protocol. By changing the request to place 1, place 0 becomes the initiator:

$$@_0 (@_1 (@_0 ((USM\ hash\ am) \sim @_1 (USM\ hash\ am))))$$

In this case, place 1 asks place 0 to hash its attestation manager while requesting that 1 measure its own attestation manager.

In a final mutual attestation example, ordering is enforced by specifying that the attestation of place 0 must occur before the attestation of place 1:

$$@_0 (@_1 (@_0 (USM\ hash\ am) \prec (USM\ hash\ am)))$$

A final example defines a protocol for the virus checking layered attestation example from the introduction. In this protocol, place 0 is the appraiser, place 1 is the target where the virus checker runs, and place 2 is the signature file server.

$$@_0((KIM\ 1\ LKIM)$$
$$\prec @_1(((USM\ hash\ vc) \sim (USM\ hash\ sf))$$
$$\sim ((KIM\ 2\ LKIM) \prec @_2 (USM\ hash\ ss))))$$

Place 0 first runs a KIM measurement on place 1. Then it asks place 1 to run a protocol that measures its virus checker, its signature file, and the signature server in parallel. Place 1 measures the signature server by first performing a KIM measurement, then requesting that place 2 measure its signature server. The KIM and place 2 measurements must occur in sequence.

Evidence from this protocol will have the form:

$$K_0^1(lkim_1) \mathbin{;;} (U_1(\#vc) \parallel U_1(\#sf) \parallel (K_1^2(lkim_2) \mathbin{;;} U_2(\#ss)))$$

where $\#x$ is the hash of x and $lkim_x$ is the result of running LKIM on place x. Evidence is appraised by: (i) checking KIM and USM evidence values against golden values; and (ii) checking measurement value composition.

3 INTERPRETING COPLAND

While the Copland language is adept at representing and verifying remote attestation protocols, it leaves many open design decisions for its surrounding execution infrastructure. A Haskell prototype demonstrates the Copland approach by providing an interpreter for protocols embedded in a full attestation manager [3]. At a high level the interpreter behaves like a canonical language interpreter; it takes a Copland protocol term and some initial evidence as input and produces an evidence value. However, the $@_p (t)$ term adds an interesting wrinkle by delegating execution of terms to remote places running their own instance of the interpreter. Thus, each interpreter is responsible not only for performing local measurements, but it must exist within a communication infrastructure to send requests and gather responses. The interpreter must also respect measurement orderings as specified by the Copland term, bundle evidence, choose cryptographic primitives, and perform appraisal.

This Haskell prototype serves as a basis for testing new Copland language extensions and as a reference interpreter as the approach is extended to other increasingly formal language environments. Two long-term goals are a collection of communicating attestation managers running on multiple operating systems in multiple language environments and formally verified interpreters for high-assurance environments. The Haskell interpreter is a first step in establishing the feasibility of the approach before verification.

Throughout interpreter and infrastructure development the architecture shown in Figure 4 has emerged with distinct logical components separate from the interpreter. The interpreter together with run-time infrastructure implements an attestation manager. While the Copland definition has been formally specified and verified, the Copland interpreter and attestation manager have not. Although this initial version will not be verified, it provides critical understanding of implementation details. Ultimately, our goal is to better understand the inherent challenges of writing and analyzing real-world remote attestation protocols.

3.1 Main Interp Loop

The main interpreter logic centers on a dispatching function, interp (Figure 5). The function is recursive and pattern matches over T(Figure 6), the Copland Haskell abstract syntax for protocol terms, invoking the corresponding measurement, crypto, or communication actions before bundling and returning an evidence package. interp also inputs initial evidence supporting evidence accumulation from earlier in protocol execution or from a prior protocol run. The top-level structure of interp mimics the evidence semantics (Figure 3) and hides much of its complexity in the USM/KIM dispatch functions (inerpUSM, interpKIM), abstract cryptographic commands (signEv, genNonce), and abstract communication actions (toRemote).

As an example of hiding implementation details, the toRemote function for $@_q (t')$ hides all underlying communication management and simply reads as a request that place q interpret Copland term t' with initial evidence e. This decoupling of high level interpreter actions from the underlying infrastructure highlights the distinction between the interpreter and the attestation manager. The interpreter provides high-level term execution derived from the semantics, while the attestation manager provides implementation details. The architecture exposes logical boundaries where we can drop in verified components as they become mature.

3.2 Places

An important design goal of the Copland interpreter is that each attestation manager instance should share as much core functionality and structure as possible, yet remain parameterized over the environment where it runs. In the Copland formal definition the abstract notion of environment is called place and is simply a natural number identifier. A good way to think of a place is an attestation manager with its own host, resources, and software configuration. Our development revealed a variety of environment-specific components associated with each place. For example, each place must manage and protect a private key. That key may take the form of a TPM Attestation Key, an RSA key pair, a PUF-based identity, or something similar to a private key in less capable (IOT) platforms [19]. The key is used for signing evidence as well as uniquely identifying the platform. The key must support identity and signing regardless of implementation, and resulting evidence must be interpreted across multiple platforms.
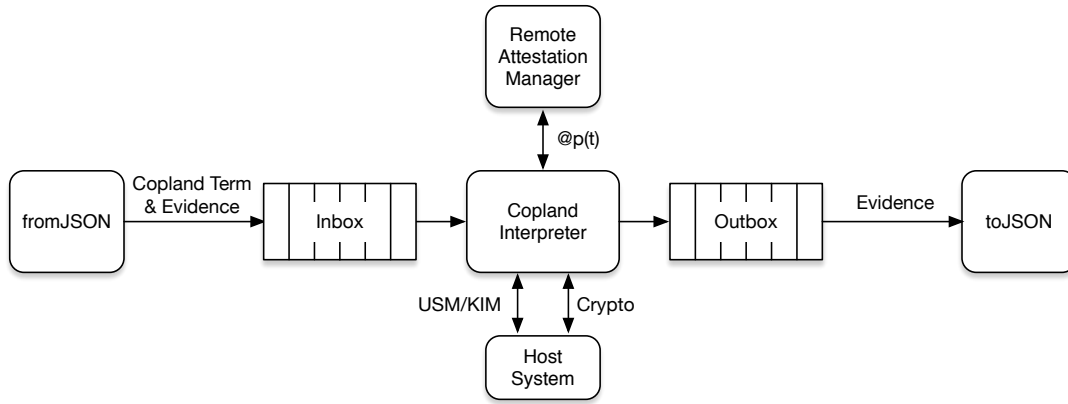
Figure 4: Attestation Manager structure and interfaces.

3.3 Signing

The `SIG` Copland term represents an abstract operation that performs a digital signature over the accumulated evidence. Interpreting `SIG` at a particular place provides authenticity and integrity meta-evidence for the evidence it consumes. In the formal specification this amounts to tagging evidence with a unique platform identifier and assuming reasonable abstract properties [4, 14] of digital signatures. However, a concrete implementation must resolve a number of details including key management and cryptographic algorithm selection. Key management is a well-known difficult problem. For this implementation we simply store keys in unprotected files and use system environment variables to point each interpreter instance to its private key. Future work will incorporate protected storage and key management with both TrustZone and TPM-based technologies. For generating keys and signing, the prototype implementation uses an eliptic-curve public-key signature system. This library is chosen for its simple API and Haskell implementation. The abstract design of our interpreter supports dropping in different algorithms to fit the security policy of each interpreter implementation.

3.4 Sequencing

Ordering is a critical part of the Copland semantics. The LN, BRS, and BRP constructors from the Haskell datatype correspond to the →, ≺, and ∼ ordering operations in the formal language, respectively. As seen in Figure 5, there are subtle differences in how `interp` must handle the various sequencing styles.

For `LN` the interpreter first evaluates `t1`, the left-most protocol subterm, using the current accumulated evidence `e` to produce the intermediate evidence value `e1`. The next line of interpretation depends on the `pseq` function from Haskell's `Control.Parallel` library that provides the user finer-grained control over evaluation order in the presence of Haskell's laziness. `pseq` takes two arguments and ensures that the first argument fully evaluates before returning the second argument. Thus, `pseq e1 (interp t2 e1)` ensures that e1 fully evaluates before returning `(interp t2 e1)`. Once `pseq` returns `(interp t2 e1)`, it can evaluate and return its result, `res`. This machinery is necessary to force sequential execution in Haskell because Haskell is a *lazy* language by default.

Without using `pseq` there is nothing preventing evaluation of the `t2` term before `e1` is fully computed.

The BRS case is similar, except for initial evidence passed to each subterm. The initial evidence `e` is first split into `es1` and `es2`. `es1` is routed to the `t1` subterm to produce evidence `e1`, and `es2` is routed to `t2`. Once again, we use `pseq` to force evaluation of `e1` before `(interp t2 es2)` can begin. Finally, we return sequential evidence via the SS constructor corresponding to ;; in the formal evidence definition.

BRP is virtually identical to BRS except the omission of `pseq`. This allows Haskell to potentially compute e1 and e2 in parallel. There are ways to explicitly inform Haskell to compute values in parallel, but this is an optimization and avoided at present. BRP constructs parallel evidence via the PP constructor corresponding to ∥ in the formal definition.

One simplification made in the Haskell implementation is restricting data splitting functions to identity and empty. The formal semantics allows arbitrary splitting functions that route initial evidence to the two constituent terms. No protocol developed thus far requires any kind of filtering or projection. Either the initial data is passed or is not passed. Thus, the current interpreter implementations only provide for identity and empty functions as splitting operations. This dramatically simplifies both verification and implementation.

The formal verification of Copland and its execution semantics relies on the interpreter having a reliable mechanism to ensure sequential measurement actions. The `pseq` solution is Haskell-specific and somewhat *ad hoc*. Attestation results are trustworthy only as much as the `pseq` implementation. This highlights a semantic gap between the formal semantics and the unverified implementation and motivates the need for such a mechanism in any instance of the interpreter. Bridging that gap will require a guaranteed sequencing operation, verified implementation, or trusted attestation manager.

3.5 Concrete Evidence

The Copland formal specification includes the definition of evidence, but the representation is held abstract and is more accurately classified as an evidence type. It describes the *structure* of the evidence without specific concrete results of measurements such as hash

```
interp :: T -> Ev -> COP Ev
interp t e = do
  p <- asks me
  ev <-
    case t of

    USM i args -> do
      bs <- interpUSM i args
      return $ U i args p bs e

    SIG -> do
      bs <- signEv e
      return $ G p e bs

    CPY -> return e

    NONCE -> do
      bs <- genNonce
      return $ N p bs e

    AT q t' -> do
      e' <- toRemote q t' e
      return e'

    LN t1 t2 -> do
      e1 <- interp t1 e
      res <- pseq e1 (interp t2 e1)
      return res

    BRS (sp1, sp2) t1 t2 -> do
      let es1 = splitEv sp1 e
      let es2 = splitEv sp2 e
      e1 <- interp t1 es1
      e2 <- pseq e1 (interp t2 es2)
      return $ SS e1 e2

    BRP (sp1, sp2) t1 t2 -> do
      let es1 = splitEv sp1 e
      let es2 = splitEv sp2 e
      e1 <- interp t1 es1
      e2 <- interp t2 es2
      return $ PP e1 e2

  return ev
```

Figure 5: Representative cases of the `interp` function.

```
type Pl = Int
type ARG = String
type ASP_ID = Int

type BS = ByteString

data T
  = USM ASP_ID [ARG]
  | KIM ASP_ID Pl [ARG]
  | SIG
  | HSH
  | NONCE
  | AT Pl T
  | LN T T
  | BRS (SP,SP) T T
  | BRP (SP,SP) T T

data Ev
  = Mt
  | U ASP_ID [ARG] Pl BS Ev
  | K ASP_ID [ARG] Pl Pl BS Ev
  | G Pl Ev BS
  | H Pl BS
  | N Pl BS Ev
  | SS Ev Ev
  | PP Ev Ev
```

Figure 6: Haskell datatypes representing Copland terms(T) and evidence (Ev).

includes places (Pl) as numbers, argument strings (ARG) and identifiers (ASP_ID) describing the USM/KIM actions performed, and raw bit strings (BS) representing results of measurement.

Although useful for formal analysis and definition purposes, place identifiers and argument strings need not be a part of cryptographic operations. To prepare evidence for hashing and signing, we extract the raw bits and compose sequential evidence by appending the raw bits recursively. There are many ways this could be done, but our initial efforts identified the importance of having a standard procedure to take evidence to a canonical representation, perform the necessary crypto, then re-package the bits into the correct location in the evidence representation. This operation is crucial to integration of remote interpreters. Our first attempt at such an operation is the encodeEv recursive Haskell function shown in Figure 7. encodeEv takes concrete evidence as input and produces raw bits as output. Note that _ in Haskell means to ignore a particular positional parameter.

3.6 Measurement

Copland defines two general measurement types. A User Space Measurement (USM) is a measurement performed in the same space as the attestation manager. A Kernel Integrity Measurement (KIM) is a measurement performed in the execution space of a different attestation manager. KIMs are used for creating trust chains. A

values, nonce values, and signatures. Every implementation must address this by introducing a concrete evidence datatype that holds actual results of protocol execution. Figure 6 shows the concrete evidence grammar, Ev, for the Haskell implementation on Linux.

By convention a concrete evidence value accumulates as Copland protocol execution proceeds. One issue encountered during development is deciding how to perform cryptographic operations over this evidence representation. Note that the evidence grammar

```
encodeEv :: Ev -> BS
encodeEv e =
  case e of
  Mt -> B.empty
  U _ _ _ bs _ -> bs
  K _ _ _ _ bs _ -> bs
  G _ _ bs -> bs
  H _ bs -> bs
  N _ bs _ -> bs
  SS e1 e2 ->
    let e1bs = (encodeEv e1) in
    let e2bs = (encodeEv e2) in
    (B.append e1bs e2bs)
  PP e1 e2 ->
    let e1bs = (encodeEv e1) in
    let e2bs = (encodeEv e2) in
    (B.append e1bs e2bs)
```

Figure 7: Cannonical way of taking concrete evidence to its binary representation.

system is trusted when it is strongly identified and directly observed behaving as expected or indirectly observed behaving as expected by a trusted third party [12]. KIM measurements that observe a different place provide both direct and indirect evidence of behavior. Once a KIM is performed, a USM may be performed by the measured system, knowing that evidence from the KIM may be appraised to determine good or bad behavior.

Copland intentionally leaves USM and KIM terms abstract in the formal definition. This allows for extending the vocabulary of measurements without extending the Copland language itself. While ideal from a language design perspective, it moves the burden of dispatching measurement routines to the interpreter on each platform. For example, consider a platform that receives a request with the following Copland term: (USM *hash* sf). Upon encountering the measurement ID *hash*, the interpreter must dispatch the measurement task to something that knows how to take a hash of the file sf.

The mapping from an abstract measurement description to a concrete implementation is called attestation policy and is implemented by the dispatcher. Each attestation manager implements policy in a manner specific to its environment. Although conceptually simple, in specifying measurement mechanism policy defines privacy. When a USM or KIM performs a measurement, policy defines what information the target is willing or able to prove to the appraiser. Conversely, when an appraiser requests a USM or KIM, policy restricts the information available. Thus, policy will be an important factor in protocol negotiation between and appraiser and target.

In the current implementation, policy is simply a Haskell function that maps well-known measurement IDs to Haskell functions that perform the measurement actions such as hashing file contents, listing a directory, or building a composite hash. Later versions of the interpreter will have USM and KIM measurement services running as dedicated servers. Nonetheless, formalizing this type of

"measurement selection" remains an open research question and moves into the realm of policy. Early experiments that involve multiple interpreters from different language/execution environments highlight the need for a centralized, precise description of each USM/KIM operation. For example, (USM *hash* sf) should take a SHA-256 hash of the raw contents of a file with an absolute pathname of sf.

3.7 Abstract Communication Layer

A distinguishing feature of remote attestation is that participants in a protocol may be on diverse platforms with drastically different capabilities and resources. An example of this in the larger context of this work is the communication infrastructure surrounding the interpreter. Some platforms have a full TCP/IP stack, others rely on VM to VM communication (XEN's vchan), others may rely on UDP, or even a custom embedded bus. An important design goal of this interpreter is making the main loop unaware of the concrete communication mechanism and interact with it via a standard send/receive interface. The interpreter should be aware when new requests for attestation have arrived and have an abstract notion of what participant sent them. It should not be concerned with how they got there or the details of finding an exact address to send the response.

We address this design goal by treating each place as a logical endpoint, each with its own mailbox for incoming messages. We leverage Haskell's Software Transactional Memory[18] (STM) library[7] to allow safe concurrent access to mailboxes, both by the communication mechanism and the interpreter loop. Not only does this abstract communication strategy support portability among diverse platforms, but it allows dropping in alternate local communication mechanisms that are ideal for testing an entire protocol execution on a single platform, using multiple threads that act as the distinct participants.

Our current implementation supports two communication modes. The first is a shared memory model where each participant has its own thread in the same process, and they communicate via shared memory. This mode is ideal for testing the interpreter logic independent of the communication infrastructure, as it requires minimal configuration and can all be orchestrated from a single platform terminal. The second mode allows for socket-based communication where each place owns an executable and listens on a designated socket for attestation requests. This mode is what a fielded attestation infrastructure should provide, but it requires additional administration that is largely independent of attestation protocol interpretation.

3.8 Appraisal

Appraisal is the process of evaluating a piece of evidence to determine if it holds the correct measurement values and is cryptographically sound. However, "correct" is a loaded term that depends on the types of measurements requested and the appraiser's standard. Recalling the virus checker example, one appraiser platform may accept a system running any one of the latest three versions of the virus checker; a more demanding appraiser may only accept the most up-to-date.

Our development efforts revealed that writing general purpose appraisers is quite difficult and raises many design questions. Among these include public key management (for checking signatures), golden measurement value management (provisioning golden hashes, storing them, etc.), nonce management, measurement policy semantics, and error reporting. In our current interpreter implementation, the appraiser is hard-coded in many places to handle a limited set of Copland protocols. In future work we hope to address the above design considerations and move towards formalization, perhaps adding appraisal primitives to the Copland language proper.

3.9 JSON Message Exchange

Because interpreters must run in diverse environments and co-ordinate with each other, it is essential that they have a shared, standard communication mechanism for Copland terms and evidence. To accomplish this the attestation manager uses the JSON message exchange format for attestation requests and responses. JSON is popular, lightweight, and we are aware of ongoing work that extends [6] to build formally verified JSON parsers.

The JSON encoding of Copland terms and evidence is a straight-forward mapping of the corresponding algebraic data types to JSON objects. Two representative examples can be seen in Figure 8. The general schema for encoding these ADTs is a JSON object with a "name" field that holds the constructor name as a JSON string. This allows a parser to distinguish a USM term and a KIM term, for example. Finally, each constructor may have arguments that are mapped to a "data" field that is a JSON array with an ordered sequence of the constructor's arguments. For recursive datatypes like the Copland sequence terms, the recursive terms are simply JSON objects themselves within the array of arguments.

While JSON provides a data exchange format, care must be taken to translate JSON structures into representations appropriate in the host language. Although Haskell is the host language discussed here, this attestation manager must eventually communicate with attestation managers running in Windows and seL4 [8], written in CakeML [9] and F#, and running on Intel and ARM processors. JSON structure processing must be done carefully and verified to ensure data formats in each host language are respected. One design decision made in the current implementation to support this is encoding raw binary data such as measurement hashes as base64-encoded JSON strings.

4 EXAMPLES

To demonstrate our interpreter and attestation manager consider the pretty-printed Haskell Copland phrase in Figure 9. This term encodes the virus checker example introduced earlier:

$$@_0 \, (\text{USM } N_0) \rightarrow @_1 \, ((\text{CPY} \overset{\pi}{\prec} (\text{USM } hash \text{ vc})) \rightarrow \text{SIG})$$

Running this term through the Haskell interpreter produces the pretty-printed final evidence result in Figure 10. An outline choosing place 0 as the outermost appraiser, interpreting this term starting with empty initial evidence, and building the final evidence value follows.

Upon encountering the outer LN term, the `interp` instance at place 0 first descends into the left subterm, here the NONCE command. Interpreting NONCE generates random bits via a Haskell library call

```
{
  "name": "AT",
  "data": [
    < number >,
    {
      "name" : <T_constructor_name>,
      "data" : [...]
    }
  ]
}
```

```
{
  "name": "K",
  "data": [
    < number >,
    [< string >],
    < number >,
    < number >,
    < string >,
    {
      "name": <Ev_constructor_name>,
      "data": [...]
    }
  ]
}
```

Figure 8: JSON schemas for AT and K.

and packages the result in the N constructor. Note that there is no NONCE command in the formal Copland language definition; we instead model nonce generation by the Copland term (USM N_0). This is an example of the role of our interpreter as a testing ground for language extensions that we hope will be integrated into the formal specification. This nonce becomes input evidence to the recursive call to `interp` for the right subterm. The right subterm here is an $@_p \, (t)$ term representing a remote request to place 1. Upon encountering the $@_p \, (t)$ term, the `toRemote` function packages the remainder of the protocol as a remote request to place 1, also passing along the nonce as initial evidence. The remote request message is shown pretty-printed in its Haskell representation in Figure 11. Besides the protocol and initial evidence terms, the request includes some bookeeping items for the communication infrastructure: destination place, source place, and a randomly-generated message id to distinguish the ensuing response message.

Upon receiving the request from place 0, place 1 begins to interpret the remainder of the protocol. Encountering the LN term, `interp` again descends to its left-most subterm, here BRS (ALL,NONE) CPY (USM 1 [''target.txt'']). The ALL evidence splitting tag routes the nonce evidence to the CPY term, while NONE routes empty evidence to the USM term. CPY simply copies the nonce evidence. USM 1 [''target.txt''] triggers the USM dispatcher to invoke the measurer with attestation service provider ID (ASP_ID) 1. The policy for place 1 maps this ASP_ID to a measurement function that hashes the contents of the file with the name "target.txt". Once CPY and USM 1 [''target.txt''] finish executing in sequence,

```
LN
  NONCE
  @_1
    LN
      BRS (ALL,NONE)
        CPY
        USM 1 ["target.txt"]
      SIG
```

Figure 9: Example Copland phrase (pretty-printed).

```
G 1
  (SS
    N 0 "\161_h\161..." (Mt)
    U 1 ["target.txt"] 1 "G\209\251\185..." (Mt))
  "\251\249a\213..."
```

Figure 10: Example Copland evidence (pretty-printed).

```
REQ "\NUL\141\..."    (* REQ constructor and message ID *)
1                     (* Destination place *)
0                     (* Source place *)
(LN                   (* Copland phrase *)
   BRS (ALL,NONE)
     CPY
     USM 1 ["target.txt"]
   SIG)
N 0 "\161_h\161..." (Mt)  (* Initial evidence *)
```

Figure 11: Request message for example protocol.

```
RES "\NUL\141\..."   (* RES constructor and message ID *)
0                    (* Destination place *)
1                    (* Source place *)
(G 1                 (* Evidence result *)
  (SS
     N 0 "\161_h\161..." (Mt)
     U 1 ["target.txt"] 1 "G\209\251\185..." (Mt))
  "\251\249a\213...")
```

Figure 12: Response message for example protocol.

the interpreter packages the result as sequential evidence. The final protocol action at place 1 is the right-most subterm of LN, here the SIG command. SIG uses the policy at place 1 to find its private key (or equivalent), choose an appropriate crypto algorithm, then generate a digital signature over the accumulated sequential evidence. Finally, place 1 sends this evidence result back to place 0 as a response message shown in Figure 12. Upon receiving the response message, place 0 bundles the final result before proceeding with appraisal or other actions on the evidence.

5 RELATED WORK

The architecture for the Copland attestation manager is derived from work by Coker Coker et al. [3]. They define an attestation architecture where protocol execution interacts with attestation

service providers (ASPs) that perform attestation services including measurement. The mapping of USM and KIM constructs in Copland performs the same function as communicating with ASPs. Furthermore, the idea of a policy mapping USM, KIM, and other functions is inspired by the same work. Copland policy as implemented is far more restricted.

The original motivation for Copland is a protocol language for the Maat [15] attestation environment. Maat provides an implementation platform for attestation protocols that serves a similar role as the Copland attestation manager presented here. While Copland is interpreted directly here, it is compiled into shell scripts that drive Maat components. Our future work will include integration with the Maat infrastructure.

Layered attestation and mutual attestation as implemented in Copland and the Copland attestation manager is motivated by Rowe [16, 17].

6 FUTURE WORK

The Haskell system presented here is the first attempt at implementing a Copland interpreter and attestation manager. The larger project extends this work to an ecosystem of formally verified attestation managers negotiating and running verified protocols on heterogeneous platforms.

6.1 Verification

The initial Copland semantics is expressed and verified in Coq. A denotational evidence semantics serves as the basis for the implementation described here. An event semantics that defines the behavior of Copland protocols provides a basis for exploring ordering among measurement operations. It also serves as a definition of a Copland intermediate form or virtual machine. Regardless, an important next step is verifying a Copland interpreter implementation.

The approach taken is to use CakeML as an implementation platform and verify an interpreter implementation in Coq. CakeML has a well-defined semantics captured in the Lem [13] semantic system. A verified compiler exists for CakeML that generates binaries for both ARM and x86 platforms from the same source with little or no modification. Producing a verified interpreter in CakeML allows us to provide correctness guarantees for the family of interpreters that follow the Haskell interpreter described here.

6.2 Target Platforms

The Haskell interpreter currently targets the RedHat and Ubuntu Linux environments. Similar capabilities are under development for Windows 10 and seL4 [8]. The current Windows 10 implementation is under development in F# and extends Copland to interact with a TPM. The rationale for this is leveraging the extensive .NET support for the TSS [20] and TPM.

The seL4 target implementation has attestation managers running both as a CAmkES [10] component and as an executable in a seL4 virtual machine. Fielding the CakeML-based Copland interpreter on seL4 provides a fully verified implementation with strong separation properties. The system may be used to host software or as a stand-along attestation capability.

6.3 Extensions

Copland defines a mechanism for performing and sequencing measurements. While useful in its current form, extensions are planned for lambda expressions and variables, managing nonces and keys, and representing stateful protocols. Lambda expressions and variables are a necessary abstraction mechanism. In addition to encapsulating reusable functionality, lambda expressions provide a powerful target for elaboration of other programming constructs such as `let` forms. Introducing state in an attestation manager makes many attestation functions simpler. Two such functions are management of nonces and keys. Currently nonces are handled in a primarily *ad hoc* fashion—they are simply random numbers treated as evidence. With a stateful environment nonce values can be maintained for simpler appraisal. Each attestation manager currently has a single key used for signing. Allowing multiple keys will allow attestation managers to provide encryption and have multiple aliases.

Extending the Copland interpreter's concept of policy is both necessary and promising. Currently, policy is a mapping of abstract measurement descriptions to concrete implementations embedded in the Copland interpreter. It defines what actually happens on the implementation platform when (`USM hash "/etc/passwd"`) executes. Policy can be much more; allowing an appraiser to negotiate protocols with a target. Currently, the appraiser provides a protocol that is executed opaquely based solely on the appraiser's identity. Providing a richer policy allows an appraiser and target to agree on the details of a protocol and evaluate protocols with respect to privacy goals. To support a richer notion of policy, future attestation managers must incorporate stateful capabilities that make decisions before, during, and after interpreting individual Copland phrases.

Finally, we are working to develop a type system for Copland terms. This type system currently considers substructural typing techniques [21] for statically checking common correctness conditions. Specifically considered are measurement ordering, nonce handling, and session key management.

7 CONCLUSIONS

Copland is a domain specific language targeting development and execution of attestation protocols. This work opens by describing the motivation for developing such a language, providing an overview of its syntax and semantics, and providing a collection of motivating examples. Next it describes the first Copland implementation as an interpreter written in Haskell. Details of an attestation manager written around Copland are described as well as mechanisms for implementing communication, measurement, and sequencing. The work closes with a description of next steps and goals of the Copland community.

ACKNOWLEDGMENTS

REFERENCES

[1] S. Berger, R. Caceres, K. Goldman, R. Perez, R. Sailer, and L. van Doorn. 2006. vTPM: Virtualizing the Trusted Platform Module. (2006). http://www.kiskeya.net/ramon/work/pubs/security06.pdf IBM T. J. Watson Research Center, Hawthorne, NY 10532 USA.

[2] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media.

[3] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. 2011. Principles of Remote Attestation. *International Journal of Information Security* 10, 2 (June 2011), 63–81.

[4] D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* 29, 2 (March 1983), 198 – 208. https://doi.org/10.1109/TIT.1983.1056650

[5] Oded Goldreich and Yair Oren. 1994. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology* 7 (1994), 1–32. Issue 1. http://dx.doi.org/10.1007/BF00195207 10.1007/BF00195207.

[6] David Hardin, Konrad Slind, Michael Whalen, and Tuan-Hung Pham. 2012. The Guardol Language and Verification System. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12).* Springer-Verlag, Berlin, Heidelberg, 18–32. https://doi.org/10.1007/978-3-642-28756-5_3

[7] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05).* ACM, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

[8] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commununications of the ACM* 53, 6 (2010), 107–115. https://doi.org/10.1145/1743546.1743574

[9] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14).* ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

[10] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. 2007. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software* 80, 5 (2007), 687–699.

[11] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. 2007. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing (STC '07).* ACM, New York, NY, USA, 21–29. https://doi.org/10.1145/1314354.1314362

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

[13] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. 2011. Lem: A Lightweight Tool for Heavyweight Semantics. In *Interactive Theorem Proving*, Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 363–369.

[14] Lawrence C Paulson. 1998. The inductive approach to verifying cryptographic protocols. *Journal of computer security* 6, 1 (1998), 85–128.

[15] J Aaron Pendergrass, Sarah Helble, John Clemens, and Peter Loscocco. 2017. Maat: A Platform Service for Measurement and Attestation. *arXiv preprint arXiv:1709.10147* (2017).

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

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

[18] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95).* ACM, New York, NY, USA, 204–213. https://doi.org/10.1145/224964.224987

[19] H. Tan, G. Tsudik, and S. Jha. 2017. MTRA: Multiple-tier remote attestation in IoT networks. In *2017 IEEE Conference on Communications and Network Security (CNS).* 1–9. https://doi.org/10.1109/CNS.2017.8228638

[20] Trusted Computing Group. 2007. *TCG TPM Specification* (version 1.2 revision 103 ed.). Trusted Computing Group, 3885 SW 153rd Drive, Beaverton, OR 97006. https://www.trustedcomputinggroup.org/resources/tpm_main_specification/

[21] David Walker. 2005. Substructural type systems. *Advanced Topics in Types and Programming Languages* (2005), 3–44.