An Infrastructure for Faithful Execution of Remote Attestation Protocols

Adam Petz(✉) and Perry Alexander

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

Abstract. Remote attestation is a technology for establishing trust in a remote computing system. Copland is a domain-specific language for specifying attestation protocols that operate in diverse, layered measurement topologies. An accompanying reference semantics characterizes attestation-relevant system events and bundling of cryptographic evidence. In this work we formally define and verify the Copland Compiler and Copland Virtual Machine for executing Copland protocols. The compiler and vm are implemented as monadic, functional programs in the Coq proof assistant and verified with respect to the Copland reference semantics. In addition we introduce the Attestation Manager Monad as an environment for managing attestation freshness, binding results of Copland protocols to variables, and appraising evidence results. These components lay the foundation for a verified attestation stack.

Keywords: Remote attestation · Verification · Domain specific languages

1 Introduction

Semantic Remote Attestation is a technique for establishing trust in a remote system by requesting *evidence* of its behavior, *meta-evidence* describing evidence properties, and locally *appraising* the result. Remote attestation by virtual machine introspection was introduced by Haldar and Franz [18] and subsequently refined [5,7,8,19,40,41] to become an important technology for security and trust establishment.

In its simplest form remote attestation involves an *attester* (or *target*) and an *appraiser*. The appraiser requests evidence from an attester that executes an *attestation protocol* sequencing *measurements* to gather evidence and meta-evidence. Upon receiving evidence from the attester, the appraiser performs an *appraisal* to determine if it can trust the attester.

This work is funded 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.

A. Dutle et al. (Eds.): NFM 2021, LNCS 12673, pp. 268–286, 2021.
https://doi.org/10.1007/978-3-030-76384-8_17

As system complexity increases so increases attestation and appraisal complexity. Federations of targets, systems-of-systems, privacy and security, and layering all introduce a need for complex, multi-party attestations. To address this need the authors and their colleagues developed Copland [39], a language for defining and executing attestation protocols. Copland has a formal semantics defining measurement, where measurement is performed, measurement ordering, and evidence bundling.

Our aspirational goal is developing a formally verified execution environment for Copland protocols. This work centers on a formal model for compiling and executing Copland in an operational environment. We define a compiler, virtual machine, and run-time environment as functional programs in Coq, then prove them compliant with the Copland formal semantics. As such it informs our development of an attestation manager in CakeML by providing a detailed formal definition of Copland protocol execution.

2 Virus Checking as Attestation

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

$$@_p\,[(\mathsf{ASP}\ vc\ \bar{a}\ p\ t)]$$

asking platform p to invoke virus checker vc as an attestation service provider targeting applications t running on p.

Simply doing a remote procedure call places full trust in vc and its operational environment. The target could lie about its results or an adversary could tamper with the virus checking system by compromising the checker or its signature file. An adversary could also compromise the operational environment running the checker or execute a man-in-the-middle replay attack.

A stronger attestation would make a request of the target that includes an encrypted nonce to ensure measurement freshness. The target would decrypt the nonce, gather evidence from the checker, and return the evidence and nonce signed using its private key. The appraiser would check the signature and nonce as well as checking the virus checker results. While the virus checker produces evidence of system state, the signature and nonce produce *meta-evidence* describing how evidence is handled. The Copland phrase for this attestation is:

$$@_p\,\{n\}[(\mathsf{ASP}\ vc\ \bar{a}\ p\ t) \rightarrow \mathsf{SIG}]$$

adding an input nonce, n, and asking p to sign the measurement result.

Evidence from the virus checker may still be compromised if the virus checker executable or signature file were compromised by an adversary. The attestation

protocol can be improved to return a measurement of the checker's operational environment in addition to virus checking results. The Copland phrase for this stronger attestation is:

$$@_p\ \{n\}[@_{ma}\ \{n\}[(\mathsf{ASP}\ h\ \bar{b}\ p\ v) \rightarrow \mathsf{SIG}] \rightarrow (\mathsf{ASP}\ vc\ \bar{a}\ p\ t) \rightarrow \mathsf{SIG}]$$

where ma is a trusted and isolated measurement and attestation domain with read access to p's execution environment. h is a composite measurement of v, the virus checking infrastructure–p's operating system along with the virus checking executable and signature file. These all occur before virus checking with the result included in a signed evidence bundle.

Measurement order is critical. An active adversary may compromise a component, engage in malice, and cover its tracks while avoiding detection. Ordering constrains the adversary by making this process more difficult [40]. If the virus checker is run before its executable or signature file are hashed the adversary has much longer to compromise the checker than if they are hashed immediately before invoking the checker. Ensuring measurement order is thus critical when verifying attestation protocols and critical to any execution or transformation of protocol representations.

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

$$@_p\ \{n\}[@_q\ \{n\}[(\mathsf{ASP}\ m\ \bar{c}\ q\ ss) \rightarrow \mathsf{SIG}] \rightarrow @_{ma}\ \{n\}[(\mathsf{ASP}\ h\ \bar{b}\ p\ v) \rightarrow \mathsf{SIG}] \rightarrow (\mathsf{ASP}\ vc\ \bar{a}\ p\ t) \rightarrow \mathsf{SIG}]$$

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

- Flexible mechanism—There is no single way for performing attestation or appraisal. A language-based approach for specifying attestation protocols is warranted [7].
- Order is important—Confidence in measurement ordering is critical to trusting an appraisal result. Preserving measurement ordering from protocol specification to execution is a critical correctness property [39–41].
- Trust is relative—Different attestations and appraisals result in different levels of trust. Formally specifying the semantics of attestation and appraisal is necessary for choosing the best protocol [7,8].

3 Copland Language and Reference Semantics

Copland is a domain specific language and formal semantics for specifying remote attestation protocols [39]. A *Copland phrase* is a term that specifies the order

and place where an attestation manager invokes attestation services. Such services include basic measurement, cryptographic bundling, and remote attestation requests. Copland is designed with expressivity and generality as foremost goals. As such Copland parameterizes attestation scenarios over work leaving specifics of measurement, cryptographic functions, and communication capabilities to protocol negotiation and instantiation.

3.1 Copland Phrases

The Copland grammar appears in Fig. 1. The non-terminal A represents primitive attestation actions including measurements and evidence operations. The constructor ASP defines an *Attestation Service Provider* and represents an atomic measurement. ASP has four static parameters, m, $\bar{a}$, p, and r that identify the measurement, measurement parameters, the place where the measurement runs, and the measurement target. A *place* parameter identifies an attestation manager environment, and supports cross-domain measurements that chain trust across attestation boundaries. Parameters to an ASP are static and must be bound during protocol selection. Protocol participants must ensure they are properly supported by the platforms involved.

$$t \leftarrow A \mid @_p\, t \mid (t \rightarrow t) \mid (t \overset{\pi}{\prec} t) \mid (t \overset{\pi}{\sim} t)$$
$$A \leftarrow \mathsf{ASP}\ m\ \bar{a}\ p\ r \mid \mathsf{CPY} \mid \mathsf{SIG} \mid \mathsf{HSH} \mid \cdots$$

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

Remaining primitive terms specify cryptographic operations over evidence already collected in a protocol run. CPY, SIG, and HSH copy, sign and hash evidence, respectively. The cryptographic implementations underlying SIG and HSH are negotiated among appraiser and target when a protocol is selected.

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

The three structural Copland terms specify the order of execution and the routing of evidence among their subterms. The phrase $t1 \rightarrow t2$ specifies that $t1$ should finish executing strictly before $t2$ begins with evidence from $t1$ consumed by $t2$. The phrase $t1 \overset{\pi}{\prec} t2$ specifies that $t1$ and $t2$ run in sequence with π specifying how input evidence is routed to the subterms. Conversely, $t1 \overset{\pi}{\sim} t2$ places

no restriction on the order of execution for its subterms allowing parallel execution. Both branching operators ($\prec$ and $\sim$) produce the product of executing their subterms.

3.2 Concrete Evidence

Copland evidence is structured data representing the result of executing a Copland phrase. Evidence and meta-evidence allow an appraiser to make a trust decision about the attesting platform. The concrete evidence definition appears in Fig. 2 and its structure corresponds closely to that of Copland phrases. Of note are the mt and N constructors that do not correspond to a Copland phrase. The former stands for "empty", or absence of evidence, and the latter for nonce evidence. Concrete measurement results are raw binary data and could be anything from a hash of software–the bs in U bs e–to a digital signature over evidene e–the bs in G bs e. The inductive e parameter accumulates sequential evidence via the $\rightarrow$ phrase, where deeper nesting implies earlier collection. Ultimately, the *guarantee* of measurement ordering comes from the Copland Virtual Machine semantics rather than the evidence structure.

$$e \leftarrow \mathsf{mt} \mid \mathsf{U}\ \mathsf{bs}\ e \mid \mathsf{G}\ \mathsf{bs}\ e \mid \mathsf{H}\ \mathsf{bs} \mid \mathsf{N}\ \mathsf{n}\ \mathsf{bs}\ e \mid \mathsf{SS}\ e\ e \mid \mathsf{PP}\ e\ e \mid \cdots$$

Fig. 2. Concrete Evidence grammar where bs is raw binary data and $\mathsf{n} = \mathit{nonce\_id} \in \mathbb{N}$

3.3 Copland LTS Semantics

The Copland framework provides an abstract specification of Copland phrase execution in the form of a small-step operational Labeled Transition System (LTS) semantics. States of the LTS correspond to protocol execution states, and its inference rules transform a Copland phrase from a protocol description to an evidence shape.

A single step is specified as $s_1 \overset{\ell}{\rightsquigarrow} s_2$ where s_1 and s_2 are states and ℓ is a label that records attestation-relevant system events. The reflexive, transitive closure of such steps, $s_1 \overset{c}{\rightsquigarrow}{}^{*}\ s_2$, collects a trace, c, of event labels representing observable attestation activity. $\mathcal{C}(t, p, e)$ represents an initial configuration with Copland phrase t, starting place p, and initial evidence e. $\mathcal{D}(p, e')$ represents the end of execution at place p with final evidence e'. Therefore, $\mathcal{C}(t, p, e) \overset{c}{\rightsquigarrow}{}^{*}\ \mathcal{D}(p, e')$ captures the complete execution of Copland phrase t that exhibits event trace c.

In addition to the operational LTS semantics, the Copland specification defines a strict partial order on attestation events called an Event System. Event Systems are constructed inductively where: (i) Leaf nodes represent base cases and hold a single event instance; and (ii) Before nodes ($t1 \triangleright t2$) and Merge nodes ($t1 \bowtie t2$) are defined inductively over terms. Before nodes impose ordering while Merge nodes capture parallel event interleaving where orderings within each sub-term are maintained. The Event System denotation function, $\mathcal{V}$, maps an

$$
\begin{aligned}
\mathcal{V}([\mathsf{SIG}]^{i}_{i+1}, p, e) &= \mathsf{SIG}_{\mathsf{event}}(i, p, [\![e]\!]_p) \\
\mathcal{V}([\mathsf{ASP}\ m\ \bar{a}\ q\ r]^{i}_{i+1}, p, e) &= \mathsf{ASP}_{\mathsf{event}}(i, p, q, r, m, \bar{a}, \mathsf{U}_{p,q,m}(e)) \\
\mathcal{V}([@_q\ t]^{i}_{j}, p, e) &= \mathsf{REQ}(i, p, q, t, e) \rhd \mathcal{V}(t, q, e) \rhd \mathsf{RPY}(j-1, p, q, \mathcal{E}(t, q, e)) \\
\mathcal{V}([t_1\ {}^{(\pi_1,\pi_2)}\!\sim\ t_2]^{i}_{j}, p, e) &= \mathsf{SPLIT}(i, (\pi_1, \pi_2), ...) \rhd \\
&\quad (\mathcal{V}(t_1, p, \pi_1(e)) \bowtie \mathcal{V}(t_2, p, \pi_2(e))) \rhd \\
&\quad \mathsf{JOIN}(j-1, ..)
\end{aligned}
$$

Fig. 3. Event system semantics (a representative subset of rules)

annotated Copland term, place, and initial evidence to a corresponding Event System. A representative subset of this semantics [39] appears in Fig. 3.

Each event instance is labeled by a unique natural number and an identifier for the place where it occurred. Measurement and cryptographic events correspond exactly to primitive Copland terms, while communication events REQ and RPY model a request and reply interaction to a remote place. The SPLIT event captures functions π_1 and π_2 that filter evidence passed to subterms, and JOIN captures the gathering of evidence from each subterm post-execution when they are combined as a composite evidence structure. Taken together, these rules are useful as a reference semantics to characterize attestation manager execution and denote evidence structure. Any valid implementation of Copland execution will obey this semantics.

4 Copland Compiler and Virtual Machine

Copland execution is implemented as a compiler targeting a monadic, virtual machine run-time. The Copland Compiler translates a Copland phrase into a sequence of commands to be executed in the Copland Virtual Machine (CVM). copland_compile (Fig. 7) takes as input an Annotated Copland term and returns a computation in the Copland Virtual Machine Monad. Annotated Copland terms extend Copland phrases with a pair of natural numbers that represent a range of identifiers. The compiler uses these ranges to assign a unique label to every system event that will occur during execution. The LTS semantics does this similarly. Event identifiers play a key role in the proof that links the LTS and CVM semantics.

The Copland Virtual Machine (CVM) Monad is a state and exception monad in Coq adapted from the Verdi framework for formally verifying distributed systems [38,47]. The CVM Monad implements the standard state monad primitives bind, return, put, and get in the cannonical way. It also provides the standard functions for executing state monad computations (runState, evalState, execState), the always-failing computation (failm), and getters/putters specialized to the CVM internal state fields. Accompanying these definitions are proofs that the CVM Monad obeys the cannonical state monad laws [15].

A general monadic computation St takes a state parameter of type S as input, and returns a pair of an optional return value of type A and an updated state. The Coq signature for St is:

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

The CVM Monad is a specialization of St with the CVM_st type as its state structure. CVM_st is a record datatype with fields that hold configuration data for the CVM as it executes.

Measurement primitives build computations in the CVM Monad that perform two primary functions: simulate invocation of measurement services and explicitly bundle the evidence results. To simulate measurement, invoke_ASP (Fig. 4) appends a measurement event to the st_trace field of CVM_st, tagging it with the parameters of the service invoked along with the unique identifier x derived from annotations on the originating ASP term. Because x is guaranteed unique per-protocol due to the way Copland terms are annotated, it can also serve as an abstract representation of the binary string measurement result. This approach accounts for multiple, independent invocations of the same ASP during a protocol and captures changes in a target's state over time. To finish, invoke_ASP bundles the result in a Copland Evidence constructor for ASPs. A single function do_prim compiles all primitive Copland terms using a similar strategy. A concrete instantiation of the CVM will require additional plumbing to map ASP_IDs and digital signature invocations to concrete measurement and cryptographic services independently validated for robustness.

```
Definition invoke_ASP (x:nat) (i:ASP_ID) (l:list Arg) : CVM EvidenceC :=
  p <- get_pl ;;
  e <- get_ev ;;
  add_tracem [Term.umeas x p i l];;
  ret (uuc i x e).
```

Fig. 4. Example monadic measurement primitive

When interpreting a remote request term $@_p t$ or a parallel branch $t1 \overset{\pi}{\sim} t2$ CVM execution relies on an external attestation manager that is also running instances of the CVM. To pass evidence to and from these external components we use the shared memory st_store component of the CVM_st, relying on glue code to manage external interaction with st_store. sendReq in Fig. 5 is responsible for placing initial evidence into the shared store at index *reqi* and initiating a request to the appropriate platform, modeled by a REQ system event. It then returns, relying on receiveResp to retrieve the evidence result after the remote place has finished execution. Uniqueness of event ids like *reqi* ensures that CVM threads will not interfere with one another when interacting with st_store.

Figure 6 shows two uninterpreted functions that simulate the execution of external CVM instances. remote_evidence represents evidence collected by running the term t at place p with initial evidence e. Similarly, remote_trace represents the list of events generated by running term t at place p. There is no evidence parameter to remote_trace because the system events generated for a term are independent of initial evidence. We provide specializations of these

```
Definition sendReq (t:AnnoTerm) (q:Plc) (reqi:nat) : CVM unit :=
  p <- get_pl ;;
  e <- get_ev ;;
  put_store_at reqi e ;;
  add_tracem [REQ reqi p q (unanno t)].
```

Fig. 5. Example monadic communication primitive

```
Definition remote_evidence (t:AnnoTerm) (p:Plc) (e:EvidenceC) : EvidenceC.

Definition remote_trace (t:AnnoTerm) (p:Plc) : list Ev.
```

Fig. 6. Primitive IO Axioms

functions for both remote and local parallel CVM instances. Because the core CVM semantics should be identical for these specializations, we also provide rewrite rules to equate them. However, their decomposition enables a smoother translation to a concrete implementation where differences in their glue code may be significant.

Each case of the Copland Compiler in Fig. 7 uses the monadic sequence operation to translate a Copland phrase into an instance of the CVM Monad over unit. The individual operations are not executed by the compiler, but returned as a computation to be executed later. This approach is inspired by work that uses a monadic shallow embedding in HOL to synthesize CakeML [20]. The shallow embedding style [16] allows the protocol writer to leverage the sequential, imperative nature of monadic notation while also having access to a rigorous formal environment to analyze chunks of code written in the monad. It also leverages Coq's built-in name binding metatheory, avoiding this notoriously difficult problem in formal verification of deeply embedded languages [1].

The first three compiler cases are trivial. The ASP term case invokes the do_prim function discussed previously that generates actions for each primitive Copland operation. The @ term case invokes sendReq, doRemote, receiveResp in sequence. sendReq was described previously and receiveResp is defined similarly. doRemote models execution of a remote CVM instance by retrieving initial evidence from the store, adding a simulated trace of remote events to st_trace, then placing the remotely-computed evidence back in the shared store. Finally, the linear sequence term $(t_1 \rightarrow t_2)$ case invokes copland_compile recursively on the subterms t_1 and t_2 and appends the results in sequence.

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

In the branch parallel case ($t_1 \overset{(sp1,sp2)}{\sim} t_2$) the commands for each subterm will execute in a parallel CVM thread. The helper function startParThreads starts threads for the two subterms then appends the trace (shuffled_events el_1 el_2) to st_trace, where el_1 and el_2 are event traces for the two subterms derived from uninterpreted functions that mimic CVM execution. shuffled_events is itself an uninterpreted function that models an interleaving of the two event traces. Event shuffling is modeled explicitly in the LTS semantics, thus we add an axiom stating that shuffled_events behaves similarly. Similar to the @ term case, we use the shared store to pass evidence to and from the parallel CVM thread for each subterm. After running both threads, we retrieve the final evidence from the result indices, combine evidence for the two subterms with a parallel evidence constructor, and emit a join event. We leave the thread model abstract in the CVM semantics so that attestation managers can run in diverse environments that may or may not provide native support for concurrency.

```
Fixpoint copland_compile (t:AnnoTerm): CVM unit :=
  match t with
  | aasp (n,_) a =>
      e <- do_prim n a ;;
      put_ev e
  | aatt (reqi,rpyi) q t' =>
      sendReq t' q reqi ;;
      doRemote t' q reqi rpyi ;;
      e' <- receiveResp rpyi q ;;
      put_ev e'
  | alseq r t1 t2 =>
      copland_compile t1 ;;
      copland_compile t2
  | abseq (x,y) (sp1,sp2) t1 t2 =>
      e <- get_ev ;;
      p <- get_pl ;;
      (e1,e2) <- split_evm x sp1 sp2 e p ;;
      put_ev e1 ;;  copland_compile t1 ;;
      e1r <- get_ev ;;
      put_ev e2 ;;  copland_compile t2 ;;
      e2r <- get_ev ;;
      join_seq (Nat.pred y) p e1r e2r
  | abpar (x,y) (sp1,sp2) t1 t2 =>
      e <- get_ev ;;
      p <- get_pl ;;
      (e1,e2) <- split_evm x sp1 sp2 e p ;;
      let (loc_e1, loc_e1') := range t1 in
      let (loc_e2, loc_e2') := range t2 in
      put_store_at loc_e1 e1 ;;
      put_store_at loc_e2 e2 ;;
      startParThreads t1 t2 p (loc_e1, loc_e1') (loc_e2, loc_e2') ;;
      (e1r, e2r) <- get_store_at_2 (loc_e1', loc_e2') ;;
      join_par (Nat.pred y) p e1r e2r
  end.

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

Fig. 7. The Copland Compiler–builds computations as sequenced CVM commands

Monadic values represent computations waiting to run. run_cvm t st (bottom of Fig. 7) interprets the monadic computation (copland_compile t) with initial

state st, producing an updated state. This updated state contains the collected evidence and event trace corresponding to execution of the input term and initial evidence in st. The evidence and event trace are sufficient to verify correctness of run_cvm with respect to the LTS semantics.

5 Verification

Verifying the Copland Compiler and Copland Virtual Machine is proving that running compiled Copland terms results in evidence and event sequences that respect the Copland reference semantics. In earlier work [39] we proved for any event v that precedes an event v' in an Event System generated by Copland phrase t ($\mathcal{V}(t,p,e) : v \prec v'$), that event also precedes v' in the trace c exhibited by the LTS semantics $\rightsquigarrow^*$. This fact is repeated here as Theorem 1, where the notation $v \ll_c v'$ means that v precedes v' in event sequence c.

Theorem 1 (LTS_Respects_Event_System). *If* $\mathcal{C}(t,p,e) \overset{c}{\rightsquigarrow}^* \mathcal{D}(p,e')$ *and* $\mathcal{V}(t,p,e) : v \prec v'$, *then* $v \ll_c v'$.

To verify the compiler and virtual machine we replace the LTS evaluation relation with executing the compiler and virtual machine and show execution respects the same Event System. Theorem 2 defines this goal:

Theorem 2 (CVM_Respects_Event_System)

If run_cvm *(*copland_compile *t)*
{ st_ev := e, st_pl := p, st_trace := [ ] } ⇓
{ st_ev := e′, st_pl := p, st_trace := c } *and*
$\mathcal{V}(t,p,e) : v \prec v'$, *then* $v \ll_c v'$.

The ⇓ notation emphasises that run_cvm is literally a functional program written in Coq. This differentiates it from the $\overset{c}{\rightsquigarrow}^*$ notation used to represent steps taken in the relational LTS semantics. run_cvm takes as input parameters a sequence of commands in the CVM Monad and a CVM_st structure that includes fields for initial evidence (st_ev), starting place (st_pl), initial event trace (st_trace), and a shared store (st_store, omitted in this theorem). It outputs final evidence, ending place, and a final trace. The first assumption of Theorem 2 states that running the CVM on a list of commands compiled from the Copland phrase t, initial evidence e, starting place p, and an empty starting trace produces evidence e' and trace c at place p. The remainder is identical to the conclusion of Theorem 1.

5.1 Lemmas

To prove Theorem 2, it is enough to prove intermediate Lemma 3 that relates event traces in the CVM semantics to those in the LTS semantics. Lemma 3 is the heart of our verification and proves that any trace c produced by the CVM semantics is also exhibited by the LTS semantics. Lemma 3 also critically proves that the CVM transforms Copland Evidence consistently with the LTS

(e_t denotes the shape of evidence e and $\mathcal{E}$ the evidence reference semantics). This allows an appraiser to rely on precise cryptographic bundling and the shape of evidence produced by a valid CVM. We can combine Lemma 3 transitively with Theorem 1 to prove the main correctness result, Theorem 2.

Lemma 3 (CVM_Refines_LTS_Event_Ordering)

If run_cvm *(*copland_compile *t)*
{ st_ev := e, st_pl := p, st_trace := [ ] } ⇓
{ st_ev := e′, st_pl := p, st_trace := c } *then*
$\mathcal{C}(t,p,e_t) \overset{c}{\rightsquigarrow}{}^{*} \mathcal{D}(p,e'_t)$ *and* $\mathcal{E}(t,p,e_t) = e'_t$

Lemma 3 rules out any "extra" CVM event traces not captured by the LTS semantics. It is worth pointing out that we could extend the CVM semantics with additional, perhaps non-attestation-relevant, system events and still prove Theorem 2 directly. This is because Theorem 2 only mentions the *ordering* of *attestation-relevant* system events captured by Event Systems. However, the indirection through the LTS semantics was a convenient refinement because of its closer compatibility with fine-grained CVM execution. The proof of Lemma 3 proceeds by induction on the Copland phrase t that is to be compiled and run through the CVM. Each case corresponds to a constructor of the Copland phrase grammar and begins by careful simplification and unfolding of run_cvm. Each case ends with applying a semantic rule of the LTS semantics.

Because we cannot perform IO explicitly within Coq, we use st_trace to accumulate a trace of calls to components external to the CVM. This trace records every IO invocation occurring during execution and their relative ordering. Lemma 4 says that st_trace is irrelevant to the remaining fields that handle evidence explicitly during CVM execution. This verifies that erasing the st_trace field from CVM_st is safe after analysis.

Lemma 4 (st_trace_irrel)

If run_cvm *(*copland_compile *t)*
{ st_ev := e, st_store := o, st_pl := p, st_trace := tr_1 } ⇓
{ st_ev := e′, st_store := o′, st_pl := p′, st_trace := _ } *and*

run_cvm *(*copland_compile *t)*
{ st_ev := e, st_store := o, st_pl := p, st_trace := tr_2 } ⇓
{ st_ev := e″, st_store := o″, st_pl := p″, st_trace := _ } *then*
$e' = e''$ *and* $o' = o''$ *and* $p' = p''$

Another key property upheld by the CVM semantics is that event traces are *cumulative*. This means that existing event traces in st_trace remain unmodified as CVM execution proceeds. Lemma 5 encodes this, saying: If a CVM state with initial trace $m \; +\!+ \; k$ interprets a compiled Copland term t and transforms the state to some new state st', and similarly t transforms a starting state with initial trace k (the suffix of the other initial trace) to another state st'', then the st_trace field of st' is the same as m appended to the st_trace field of st''. We proved this vital "distributive property" over traces and leveraged it in several other Lemmas to simplify event insertion and trace composition.

Lemma 5 (st_trace_cumul)

If run_cvm *(*copland_compile *t)*
{ st_ev := e, st_store := o, st_pl := p, st_trace := m ++ k } ⇓ *st′ and*

run_cvm *(*copland_compile *t)*
{ st_ev := e, st_store := o, st_pl := p, st_trace := k } ⇓ *st″ then*
*(*st_trace *st′)* = *m* ++ *(*st_trace *st″)*

5.2 Automation

There are many built-in ways to simplify and expand expressions in Coq. Unfortunately, it is easy to expand either too far or not enough. The Coq cbv (call-by-value) tactic unfolds and expands as much as it can, often blowing up recursive expressions making them unintelligible. The milder cbn (call-by-name) tactic often avoids this, but fails to unfold user-defined wrapper functions. To reach a middle-ground, we define custom automation in Ltac. First we define a custom "unfolder" that carefully expands primitive monadic operations like bind and return, along with CVM-specific helper functions like invoke_ASP.

Next we define a larger simplifier that repeatedly invokes the targeted unfolder followed by cbn and other conservative simplifications. This custom simplification is the first step in most proofs and is repeated throughout as helper Lemmas transform the proof state to expose more reducible expressions. We also leveraged the StructTact [48] library, a collection of general-purpose automation primitives for common Coq structures like match and if statements, originally developed for use in the Verdi [38,47] framework. Combined with our custom automation this made our proofs robust against small changes to the CVM implementation (i.e. re-naming/re-ordering monadic helper functions), and greatly simplified proof maintenance after more significant refactoring.

Lemma 6 (abpar_store_non_overlap)

If well_formed *(abpar _ _ t_1 t_2) and range t_1 = (a,b) and range t_2 = (c,d) then $a \neq c$ and $b \neq c$ and $b \neq d$*

A final custom piece of automation involves Lemmas that ensure accesses to the shared store do not overlap when interpreting Copland terms that interact with external components. When compiling the branch parallel term we derive indices from term annotations and use them to insert initial evidence and retrieve final evidence from the store. We must prove arithmetic properties like Lemma 6 to show that store accesses do not overlap. The proof follows from the definition of the well_formed predicate and the annotation strategy. We provide Ltac scripts to recognize proof states that are blocked by store accesses and discharge them using Lemma 6.

6 Attestation Manager(AM) Monad

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

An early prototype of the AM Moand in Haskell [37] uses monad transformers to provide a sufficient computational context for attestation and appraisal. An example pseudo-code sequence of AM Monad commands appears in Fig. 8. The run_cvm(t, n) command runs an entire Copland phrase t with initial evidence n inside the CVM Monad, lifting its evidence result into the AM Monad. Rather than performing measurements directly, the AM Monad relies on run_cvm as a well-defined interface to the CVM. This allows an AM to abstract away details of Copland phrase execution and compose facts about the CVM like those verified in Sect. 5 about events and evidence shapes. An initial formal definition of the AM Monad in Coq, including nonce management and Copland phrase invocation, is complete. The design of appraisal and its verification are ongoing.

```
let t = @42 (ASP 1 ā p r → SIG)
  n ← generate_nonce
  e ← run_cvm(t,n)
  b ← appraise(t,e)
  trust_decision(b)
```

Fig. 8. Example sequence of commands in the AM Monad.

Using nonces is a common mechanism for preventing a man-in-the-middle adversary from re-transmitting stale measurements that do not reflect current system state. Nonces are critical to attestation and appear in Copland as primitive evidence. Since evidence collection is cumulative in the CVM semantics, nonces are generated and stored in the AM Monad state, embedded as initial evidence alongside Copland attestation requests, then retrieved during appraisal.

Appraisal is the final step in a remote attestation protocol where an indirect observer of a target platform must analyze evidence in order to determine the target's trustworthiness. Regardless of its level of scrutiny, an appraiser must have a precise understanding of the structure of evidence it examines. The Copland framework provides such a shared evidence structure, and Copland phrases executed by the CVM produce evidence with a predictable shape. The AM Monad provides an ideal context to perform appraisal because it can access golden measurement and nonce values, cryptographic keys, and also link evidence to the Copland phrase that generated it. This combination of capabilities enables automatic synthesis of appraisal routines left for future work.

7 Related Work

Integrity measurement tools include both static [29,42] and dynamic [10,17,21, 22,43,49] approaches that support both baseline and recurring measurements of target systems. More general frameworks [27,31,35] support higher-level attestation goals by combining primitive attestation services. Of note is the Maat framework [31] that introduced the term *Attestation Service Provider* and motivated the design of the Copland language. Other tools provide more specialized measurement capabilities like userspace monitoring [14,32], VMM [13,46] and kernel-level [28,33,34] introspection, and attestation of embedded/IoT platforms [3,6,25,44,45].

The framework presented in the current work is designed as a complimentary operational environment for the above tools. For more general frameworks like Maat, we envision invoking sequences of ASPs within their environment as a service (and vice-versa). For the more specialized measurement tools, we can plug them in as primitive ASPs and compose their results as Copland Evidence.

Prior work in analysis of remote attestation systems involves virtualized environments [2,7,27], comparing protocol alternatives [40,41], and semantics of attestation [9,12]. These analyses articulate the complexities in the attestation design space and lay a foundation for future frameworks. Coker et al. [7] is of particular influence, as the design principle of Trustworthy Mechanism was a primary motivation of this work.

HYDRA [11] (Hybrid Design for Remote Attestation) was the first hardware/software hybrid RA architecture to build upon formally-verified components, and that achieved design goals laid out in their prior work [12]. ERASMUS [4] levereged HYDRA as a base security architecture, but added real-time assurances for resource-critical embedded platforms. VRASED (Verifiable Remote Attestation for Simple Embedded Devices) extended these ideas to a concrete RA design, becoming the first formal verification "of a HW/SW co-design implementation of any security service" [30]. They verify end-to-end security and attestation soundness properties in LTL by automatically extracting hardware properties from Verilog specifications and manually incorporating independently-verified cryptographic software properties. While their end-to-end security guarantees are convincing for a specific embedded platform, our attestation managers support a wider range of attestation scenarios.

8 Future Work and Conclusion

In this work we have verified the Coq implementation of a Copland compiler and monadic virtual machine. Specifically, we proved that the output of compilation and virtual machine execution respects the small-step LTS Copland semantics. Artifacts associated with this verification are publicly available on github [36]. All proofs are fully automated and the only admitted theorems are axioms that model interaction with IO components external to the core virtual machine.

Verification of the compiler and vm are part of our larger effort to construct a formally verified attestation system. Figure 9 shows this work in context with

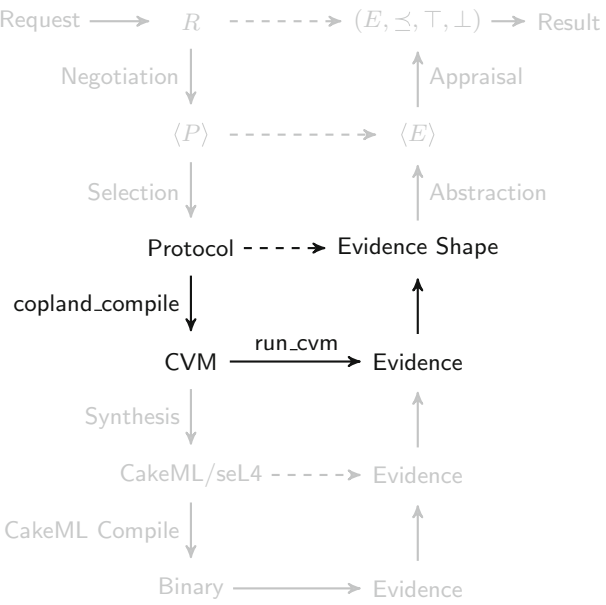

Fig. 9. Verification stack showing verification dependencies and execution path. Solid lines represent implementations while dashed lines represent mathematical definitions.

gray elements representing supporting work or work in progress. Above protocol execution is a negotiation process that selects a protocol suitable to both appraiser and target. Ongoing work will formally define a "best" protocol and verify the negotiated protocol is sufficient and respects privacy policy of all parties.

Below protocol execution is an implementation of the Copland Compiler and Copland Virtual Machine in CakeML [26] running on the verified seL4 microkernal [23,24]. CakeML provides a verified compilation path from an ML subset to various run-time architectures while seL4 provides separation guarantees necessary for trusted measurement. We are embedding the semantics of CakeML in Coq that will in turn be used to verify the compiler and vm implementations. Unverified implementations of both components have been implemented and demonstrated as a part of a hardened UAV flight control system.

When completed our environment will provide a fully verified tool stack that accepts an attestation request, returns evidence associated with that request, and supports sound appraisal of that evidence. Analysis tools that compare protocol alternatives will benefit from implementations that are faithful to formal artifacts, ultimately enabling more robust trust decisions. This work is an important first step creating a verified operational environment for attestation.

References

1. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 3–15. ACM, New York (2008). https://doi.org/10.1145/1328438.1328443
2. Berger, S., Caceres, R., Goldman, K., Perez, R., Sailer, R., van Doorn, L.: vTPM: Virtualizing the Trusted Platform Module. iBM T. J. Watson Research Center, Hawthorne (2006). http://www.kiskeya.net/ramon/work/pubs/security06.pdf
3. Brasser, F., El Mahjoub, B., Sadeghi, A.R., Wachsmann, C., Koeberl, P.: Tytan: tiny trust anchor for tiny devices. In: Proceedings of the 52nd Annual Design Automation Conference. DAC 2015. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2744769.2744922
4. Carpent, X., Rattanavipanon, N., Tsudik, G.: ERASMUS: efficient remote attestation via self- measurement for unattended settings. CoRR abs/1707.09043 (2017). http://arxiv.org/abs/1707.09043
5. Challener, D., Yoder, K., Catherman, R.: A Practical Guide to Trusted Computing. IBM Press, Indianapolis (2008)
6. Clemens, J., Pal, R., Sherrell, B.: Runtime state verification on resource-constrained platforms. In: MILCOM 2018–2018 IEEE Military Communications Conference (MILCOM), pp. 1–6 (2018)
7. Coker, G., et al.: Principles of remote attestation. Int. J. Inf. Secur. **10**(2), 63–81 (2011)
8. Coker, G.S., Guttman, J.D., Loscocco, P.A., Sheehy, J., Sniffen, B.T.: Attestation: evidence and trust. In: Proceedings of the International Conference on Information and Communications Security, vol. LNCS 5308 (2008)
9. Datta, A., Franklin, J., Garg, D., Kaynar, D.: A logic of secure systems and its application to trusted computing. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 221–236. IEEE (2009)
10. Davi, L., Sadeghi, A.R., Winandy, M.: Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In: Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing, STC 2009, pp. 49–54. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1655108.1655117
11. Eldefrawy, K., Rattanavipanon, N., Tsudik, G.: Hydra: hybrid design for remote attestation (using a formally verified microkernel). In: Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2017, pp. 99–110. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3098243.3098261
12. Francillon, A., Nguyen, Q., Rasmussen, K.B., Tsudik, G.: A minimalist approach to remote attestation. In: 2014 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1–6 (2014)
13. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: NDSS (2003)
14. Gevargizian, J., Kulkarni, P.: Msrr: measurement framework for remote attestation. In: 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), Dependable, Autonomic and Secure Computing (DASC 2018), pp. 748–753 (2018)

15. Gibbons, J.: Unifying theories of programming with monads. In: Wolff, B., Gaudel, M.-C., Feliachi, A. (eds.) UTP 2012. LNCS, vol. 7681, pp. 23–67. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35705-3_2
16. Gill, A.: Domain-specific languages and code synthesis using Haskell. Commun. ACM **57**(6), 42–49 (2014). https://doi.org/10.1145/2605205, also appeared in ACM Queue **12**(4) (2014)
17. Gopalan, A., Gowadia, V., Scalavino, E., Lupu, E.: Policy driven remote attestation. In: Prasad, R., Farkas, K., Schmidt, A.U., Lioy, A., Russello, G., Luccio, F.L. (eds.) MobiSec 2011. LNICST, vol. 94, pp. 148–159. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30244-2_13
18. Haldar, V., Chandra, D., Franz, M.: 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 (2004)
19. Halling, B., Alexander, P.: Verifying a privacy CA remote attestation protocol. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 398–412. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_27
20. Ho, S., Abrahamsson, O., Kumar, R., Myreen, M.O., Tan, Y.K., Norrish, M.: Proof-producing synthesis of cakeml with I/O and local state from monadic HOL functions. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Automated Reasoning - 9th International Joint Conference (IJCAR). Lecture Notes in Computer Science, vol. 10900, pp. 646–662. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-94205-6_42, https://cakeml.org/ijcar18.pdf
21. Jaeger, T., Sailer, R., Shankar, U.: Prima: Policy-reduced integrity measurement architecture. In: Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies, SACMAT 2006, pp. 19–28. Association for Computing Machinery, New York (2006). https://doi.org/10.1145/1133058.1133063
22. Kil, C., Sezer, E.C., Azab, A.M., Ning, P., Zhang, X.: Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In: 2009 IEEE/IFIP International Conference on Dependable Systems Networks, pp. 115–124 (2009). https://doi.org/10.1109/DSN.2009.5270348
23. Klein, G., et al.: sel4: formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010). https://doi.org/10.1145/1743546.1743574
24. Klein, G., et al.: sel4: formal verification of an os kernel. In: SOSP 2009: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 207–220. ACM, New York (2009). https://doi.org/10.1145/1629575.1629596
25. Koeberl, P., Schulz, S., Sadeghi, A.R., Varadharajan, V.: Trustlite: a security architecture for tiny embedded devices. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys 2014. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2592798.2592824
26. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: a verified implementation of ml. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,POPL 2014, pp. 179–191. ACM, New York (2014). , https://doi.org/10.1145/2535838.2535841
27. Lauer, H., Salehi, S.A., Rudolph, C., Nepal, S.: User-centered attestation for layered and decentralised systems. Workshop on Decentralized IoT Security and Standards (DISS) (2018)
28. Loscocco, P.A., Wilson, P.W., Pendergrass, J.A., McDonell, C.D.: Linux kernel integrity measurement using contextual inspection. In: Proceedings of the 2007 ACM workshop on Scalable trusted computing, STC 2007, pp. 21–29. ACM, New York (2007). https://doi.org/10.1145/1314354.1314362

29. Maliszewski, R., Sun, N., Wang, S., Wei, J., Qiaowei, R.: Trusted boot (tboot) (2). http://sourceforge.net/p/tboot/wiki/Home/
30. Nunes, I.D.O., Eldefrawy, K., Rattanavipanon, N., Steiner, M., Tsudik, G.: Vrased: a verified hardware/software co-design for remote attestation. In: Proceedings of the 28th USENIX Conference on Security Symposium, SEC 2019, pp. 1429–1446. USENIX Association, USA (2019)
31. Pendergrass, J.A., Helble, S., Clemens, J., Loscocco, P.: A platform service for remote integrity measurement and attestation. In: MILCOM 2018–2018 IEEE Military Communications Conference (MILCOM), pp. 1–6 (2018). https://doi.org/10.1109/MILCOM.2018.8599735
32. Pendergrass, J.A., et al.: Runtime detection of userspace implants. In: MILCOM 2019–2019 IEEE Military Communications Conference (MILCOM), pp. 1–6 (2019). https://doi.org/10.1109/MILCOM47813.2019.9020783
33. Petroni, N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 103–115. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1315245.1315260
34. Petroni Jr, N., Fraser, T., Walters, A., Arbaugh, W.: An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: Proceedings of the 15th USENIX Security Symposium, pp. 289–304 (2006)
35. Petz, A., Alexander, P.: A copland attestation manager. In: Hot Topics in Science of Security (HoTSoS 2019), Nashville, TN (2019)
36. Petz, A.: copland-avm, nfm21 release (2020). https://github.com/ku-sldg/copland-avm/releases/tag/v1.0
37. Petz, A., Komp, E.: haskell-am (2020). https://github.com/ku-sldg/haskell-am
38. Plse, U.: Verdi (2016). https://github.com/uwplse/verdi
39. Ramsdell, J., et al.: Orchestrating layered attestations. In: Principles of Security and Trust (POST 2019), Prague, Czech Republic (2019)
40. Rowe, P.D.: Confining adversary actions via measurement. In: Third International Workshop on Graphical Models for Security, pp. 150–166 (2016)
41. Rowe, P.D.: Bundling evidence for layered attestation. In: Franz, M., Papadimitratos, P. (eds.) Trust 2016. LNCS, vol. 9824, pp. 119–139. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45572-3_7
42. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementatation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th USENIX Security Symposium. USENIX Association, Berkeley (2004)
43. Shi, E., Perrig, A., Van Doorn, L.: Bind: A fine-grained attestation service for secure distributed systems. In: 2005 IEEE Symposium on Security and Privacy, pp. 154–168. IEEE (2005)
44. Tan, H., Tsudik, G., Jha, S.: Mtra: multiple-tier remote attestation in IoT networks. In: 2017 IEEE Conference on Communications and Network Security (CNS), pp. 1–9 (2017). https://doi.org/10.1109/CNS.2017.8228638
45. Wedaj, S., Paul, K., Ribeiro, V.J.: Dads: decentralized attestation for device swarms. ACM Trans. Priv. Secur. **22**(3), 19:1–19:29 (2019). https://doi.org/10.1145/3325822
46. Wei, J., Pu, C., Rozas, C.V., Rajan, A., Zhu, F.: Modeling the runtime integrity of cloud servers: a scoped invariant perspective. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, pp. 651–658 (2010). https://doi.org/10.1109/CloudCom.2010.29

47. Wilcox, J.R., et al.: Verdi: a framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 357–368. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2737924.2737958
48. Woos, D., Wilcox, J.R., Simmons, K., Palmskog, K., Doenges, R.: Structtact coq library (2020). https://github.com/uwplse/StructTact
49. Xu, W., Ahn, G.-J., Hu, H., Zhang, X., Seifert, J.-P.: DR@FT: efficient remote attestation framework for dynamic systems. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 182–198. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15497-3_12