Cryptographically Assured Information Flow: Assured Remote Execution*

Scott L. Dyer, Christian A. Femrite,
Joshua D. Guttman**, Julian P. Lanson, Moses D. Liskov

The MITRE Corporation

Abstract. *Assured Remote Execution* for a device is the ability of suitably authorized parties to construct secure channels with known processes (i.e. processes executing known code) running on that device. Assured remote execution requires a hardware basis including cryptographic primitives.

We show here that a simple hardware-level mechanism called *Cryptographically Assured Information Flow* (CAIF) enables Assured Remote Execution. CAIF is akin to existing Trusted Execution Environments, but securely implements an ideal functionality defined in terms of *logging* and *confidential escrow*.

We show how to achieve Assured Remote Execution for a wide variety of processes on a CAIF device. Cryptographic protocol analysis demonstrates our security goals are achieved even against a strong adversary that may modify our programs and execute unauthorized programs on the device.

Assured remote execution allows trustworthy remote attestation, and, in part, secure remote reprogramming.

1 Introduction

Suppose you have control of a device d early in its life, after which d may be physically inaccessible, e.g. on a satellite, or rarely accessible, e.g. one of many devices on ships, or embedded in airplanes, or scattered throughout the electric power system. Long-term, can you deliver messages exclusively to specific, known processes executing on d? Can you, when receiving certain messages, be sure they were prepared by specific, known processes on d? Can these processes run code written and delivered long after d was initialized?

This is the *Assured Remote Execution* challenge; we identify the known processes by the hash of their executable code.

Assured remote execution requires hardware support, as well as cryptography to protect messages in transit and to ensure authenticity of the endpoint d and active process within d. Good solutions should:

* Copyright © 2024, The MITRE Corporation. All rights reserved.

** Corresponding author, email address `guttman@mitre.org`. An extended version of this article is available at https://arxiv.org/abs/2402.02630.

1. Use a simple hardware basis relying only on simple, efficient, well-understood crypto primitives;
2. Achieve the assured remote execution even against a strong adversary capable of running its own software on the device, or modifying existing software, including hypervisor software and software running during boot;
3. Yield a verification strategy for using the mechanism, including the assured remote execution protocols.

We define here a hardware basis adapted from existing Trusted Execution Environments. It uses cryptography to satisfy an ideal functionality controlling information flow among processes local to d. We call our design CAIF, for *Cryptographically Assured Information Flow*.

Our hardware basis uses only key derivation functions, message authentication codes (MACs), and authenticated symmetric encryption. These form a small collection of deeply understood primitives, meeting criterion 1.

We have designed our mechanisms under the assumption that some processes on our devices may run carefully vetted, trustworthy code, whereas others may run questionable or even malicious code. We do not assume protected storage to hold executables within a CAIF device, so our conclusions hold even if an adversary modifies our programs in the filesystem, meeting criterion 2.

So CAIF can help software at higher level prevent malicious execution without circular dependencies.

We characterize CAIF by an ideal functionality [18,11], proving lemmas saying the ideal functionality enforces our intended information flow. We then introduce a concrete cryptographic mechanism. We use standard cryptographic proofs to show that CAIF simulates the ideal functionality except with negligible probability (Cor. 1), and thus satisfies computational approximations of the lemmas.

Having justified the CAIF mechanism, we show how to build assured remote execution on top of it. We formalize the behaviors as protocols in CPSA, a symbolic-style Cryptographic Protocol Shapes Analyzer that supports both message passing and local device state [39]. CPSA helped us eliminate errors, discover core ideas, and assure that the resulting mechanisms satisfy our security claims. Our CPSA models incorporate a strong adversary that can run any code, subject to the assumption that code that yields the same hash value under a strong hash function will also yield the same computational behavior. The ideal functionality proof and symbolic protocol analysis together meet criterion 3.

The core idea of CAIF. CAIF provides two central functions. One enables a *service* (certain processes) to *log* itself as the source or authorizer of a data item. Other active parties can subsequently query or *check* whether an expected service has logged a data item. The other central function enables a service to *escrow* a piece of data to be delivered to a particular service as recipient. Only that recipient can then *retrieve* the data value, but only if its expected source is the true source.

The cryptography-free *ideal functionality* of Section 2 implements these two parts via an *unbounded secure memory*, proving desirable behavioral properties

(Lemmas 1–3). The ideal functionality uses its unbounded secure memory to hold the logged associations of service and data, and the escrowed associations of data, source, and recipient.

CAIF devices (Section 3) use key derivation with MACs for logging and authenticated encryption for data escrow. They require little secure storage, namely a single unshared secret seed or "intrinsic secret" IS, as an input to key derivation.

The quantum resistant transition. CAIF's long-term guarantees are independent of asymmetric cryptography such as digital signatures. Motivated by the quantum cryptanalytic threat, new quantum-resistant primitives are now in draft standard [2,35,36,37]. However, the guarantees on long-lived CAIF devices are unaffected if these primitives are broken and revised, or if their key sizes must be adjusted.

New asymmetric algorithms can be installed securely on geographically dispersed CAIF devices. Indeed, CAIF shows that a long-term security architecture can depend only on stable, efficient symmetric cryptographic primitives.

Trusted Execution Environments. If all asymmetric algorithms may evolve, existing Trusted Execution Environments (TEEs) [13,23,24] are not the right tool. Although they use only symmetric cryptography at the hardware level, they rely on public key encryption to protect data being passed from one enclave to another. This was acceptable when the quantum cryptanalysis threat seemed distant, but is no longer.

CAIF contrasts in two central ways with prior TEEs. First, prior TEEs construct a device-local secret unique to each enclave, and deliver this to the enclave for actions including local attestations. CAIF constructs such a key but instead itself generates MAC tags to log data. The TEE behavior does not satisfy a logging ideal functionality. Second, CAIF's escrow behavior also constructs keys for ordered *pairs* of services, and uses those for its confidential escrows. This provides a purely symmetric method for service-to-service information flow on a device, which existing TEEs entirely lack (see Section 8.1).

Contributions. We make three main contributions.

1. We define CAIF and its ideal functionality, including the new escrow behavior.
2. We prove that CAIF, implemented with strong cryptography, is computationally indistinguishable from an instance of the ideal functionality (Section 4).
3. We develop a sequence of protocols on top of CAIF. We use symbolic protocol analysis to demonstrate that they achieve assured remote execution despite a strong adversary that can execute code of its own choice on our devices. This is the subject of Sections 5–7.

Symbolic protocol analysis is reliable here, because Cor. 1 shows that cryptography as used in CAIF yields a functionality indistinguishable from logging and escrow. Our symbolic modeling is based on this crypto-free ideal functionality. Section 5 provides an overview of our strategy for achieving assured remote execution.

Our guarantees are independent of delicate systems-level considerations, such as how software obtains control at boot.

Sections 8 and 9 discuss related work and conclude.

Terminology. We write our hardware-based symmetric mechanisms as $\mathsf{kdf}_h(x)$, $\mathsf{mac}_h(k, v)$, and $\mathsf{enc}_h(v, k)$ for the hardware key derivation function, MAC, and authenticated encryption. The corresponding hardware decryption is $\mathsf{dec}_h(v, k)$.

We write $[\![v]\!]_k$ for a digital signature on message v with signing key k. We will assume that v is recoverable from $[\![v]\!]_k$. The latter could be a pair $(v,\ \mathsf{dsig}(\mathsf{hash}(v),\ k))$ where dsig is a digital signature algorithm.

A lookup table (e.g. a hash table) is a set T of index-to-result mappings. Each mapping takes the form *index* $\mapsto$ *result*. A table T satisfies the "partial function" constraint: if $i \mapsto r \in T$ and $i \mapsto r' \in T$, then $r = r'$. T's domain is $\mathsf{dom}(T) = \{i\colon \exists r\,.\, i \mapsto r \in T\}$, and $\mathsf{ran}(T) = \{r\colon \exists i\,.\, i \mapsto r \in T\}$.

When D is a distribution, we write $\mathsf{supp}(D)$ for its support, i.e. $\mathsf{supp}(D) = \{x\colon 0 < Pr[y \leftarrow D; y = x]\}$.

2 An Ideal Functionality for CAIF

We start with an *ideal functionality* IF, meaning a well-defined set of behaviors that might be difficult to achieve directly. IF would require an unbounded amount of memory under its exclusive control, about which no observer gains any information except through IF's official interface. Lemmas 1–3 prove desirable behavioral properties of the IF.

In Section 3 we will introduce CAIF devices using cryptography, and in Section 4 we prove that these CAIF devices offer a near approximation to the IF's behavioral properties.

2.1 Main elements

We consider a system as a collection of *active processes* that act by executing instructions. Some active processes are distinguished as *services*. A service has an unchanging executable code segment, and an unshared heap for private computations. Because the code segment is unchanging, its contents serve as a persistent *principal* or *identity* for the service, and also determines its computational behavior.

The instruction set includes two pairs of special instructions besides normal computational steps. The first pair allows a service to *log* itself in an attestation log atlog as source or authority for data, so other active processes can later make decisions based on its provenance:

iattest has one parameter, which points to a region of data with some contents v. The logging functionality selects a *tag*, a bitstring τ, and stores a record into atlog associating the currently active service identity P_s with v and the tag τ.

The logging functionality returns τ in response.

icheck has three parameters, namely a service principal identity P_s, a pointer to a region of data with some contents v, and a tag τ. The logging functionality returns *true* if the named service P_s has previously logged v into atlog via iattest, and received the tag τ.

Any active process can use icheck to see if the service P_s has logged itself as an authority for v. However, only a service can execute iattest, since only services have a persistent identity P_s. One implements this functionality using a MAC, with τ selected as the MAC tag. The choice of τ is determined by some function F_{log} of P_s and v.

The second pair of instructions ensures provenance of the source, and also provides data *escrow* through a table protstore, meaning that the source service P_s is making the data v available to one recipient service P_r:

iprotect has two parameters, the intended recipient service principal identity P_r and a pointer to a region of data with some contents v. When executed by a currently active service identity P_s, the escrow functionality selects a *handle* η, and stores a record in the lookup table protstore, indexed by (η, P_s, P_r), pointing to the value v. We write this $(\eta, P_s, P_r) \mapsto v$.
The logging functionality returns η in response.

iretrieve has two parameters, the expected source service principal identity P_s and a handle η. When executed by a currently active service identity P_r, the escrow functionality does a lookup in the table protstore for index (η, P_s, P_r). If any entry $(\eta, P_s, P_r) \mapsto v$ is present in protstore, that v is returned to P_r.

Authenticated symmetric encryption is natural here. The handle η is a ciphertext, and the "authenticated" property ensures that a retrieved v was in fact previously protected by the source P_s for the recipient P_r (see Section 3).

2.2 Behavioral lemmas about IF

A strength of the ideal functionality definition is that several properties of the *behaviors* of an IF follow easily from it. A *command* c is an instruction together with a choice of its command arguments, v for iattest; (P_s, v, τ) for icheck; etc.

Definition 1. *An* event *is a triple (command,principal,result) of: a command; the executing service principal that causes it (or $\perp$ if the active process is not a service); and the result of executing the instruction.*

A behavior *of a state machine M equipped with principal identities is a finite sequence $\langle (c_i, P_i, r_i) \rangle_{i<\ell}$ of events in which each command c_i can cause the result r_i when executed by principal P_i in some state that can arise from the preceding events $\langle (c_j, P_j, r_j) \rangle_{j<i}$, starting from an initial state.*

An IF behavior *is a behavior of* IF *starting from the initial state with empty lookup tables.* ///

Properties of IF: Logging. We can summarize the important properties of the attestation instructions in a lemma. It says that a check after a matching attest does yield true, and that if a check yields true, then an earlier attest occurred.

Lemma 1 **($\mathsf{iattest}$ log correctness).** *Let $\alpha = \langle (c_i, P_i, r_i) \rangle_i$ be an* IF *behavior.*

1. *If, for $i < j$, $c_i = \mathsf{iattest}(v)$ and $c_j = \mathsf{icheck}(P_i, v, r_i)$, then $r_j = true$.*
2. *If $c_j = \mathsf{icheck}(p, v, \tau)$ and $r_j = true$, then for some $i < j$, $c_i = \mathsf{iattest}(v)$, $P_i = p$, and $r_i = \tau$.* ///

This lemma is independent of how F_{log} chooses tags.

Properties of IF: Protection. A claim like Lemma 1 holds for $\mathsf{iprotect}$ and $\mathsf{iretrieve}$, for essentially the same reasons:

Lemma 2 **($\mathsf{iprotect}$ log correctness).** *Let $\alpha = \langle (c_i, P_i, r_i) \rangle_i$ be an* IF *behavior.*

1. *If, for $i < j$, $c_i = \mathsf{iprotect}(P_j, v)$ and $c_j = \mathsf{iretrieve}(P_i, r_i)$, then $r_j = v$.*
2. *If $c_j = \mathsf{iretrieve}(p, \eta)$ and $r_j = v$, then for some $i < j$, $c_i = \mathsf{iprotect}(P_j, v)$, $P_i = p$, and $r_i = \eta$.* ///

Ideal secrecy for IF. The instructions of IF leak "no information" about the values associated with handles that are never retrieved, if we are a bit more specific about how IF chooses the handles η in $\mathsf{iprotect}$. We assume

(i) IF uses a distribution D_{j,P_s,P_r} choosing handles η for data of length j escrowed by principal P_s for recipient P_r; the choice of η being independent of which v is presented, for all v of the same length j.

(ii) The IF does not re-use any handle η for new commands; it checks the table entries and re-samples as needed.

(iii) If the lengths $j \neq j'$, then the supports $\mathsf{supp}(D_{j,P_s,P_r})$ and $\mathsf{supp}(D_{j',P_s',P_r'})$ are disjoint.

A *schematic behavior* α_ν in the variable ν results from a behavior α by replacing one occurrence of a bitstring in a command or result of α with the variable ν. If b is any bitstring, $\alpha_\nu[b/\nu]$ is the result of replacing the occurrence of ν by b. The latter may not be a behavior at all, since this b may be incompatible with other the events in α.

By (i), a strong, Shannon-style perfect secrecy claim holds for the ideal functionality:

Lemma 3 **(Perfect secrecy for $\mathsf{iprotect}$).** *Let $\alpha_\nu = \langle (c_i, P_i, r_i) \rangle_i$ be a schematic behavior, where ν occurs in an $\mathsf{iprotect}$ instruction $c_i = \mathsf{iprotect}(P_r, \nu)$ executed by P_s. Let ℓ be a length of plaintexts for which the result r_i is possible. By assumption (iii), there is a unique such ℓ. Let D be any distribution with $\mathsf{supp}(D) \subseteq \{0,1\}^\ell$.*

Suppose there is no subsequent $c_j = \mathsf{iretrieve}(P_i, r_i)$ with this input r_i and $P_j = P_r$. For every $b \in \mathsf{supp}(D)$:

1. *$\alpha_\nu[b/\nu]$ is a behavior;*

2. *the probability* $Pr[v_0 \leftarrow D; v_0 = b \mid \alpha_\nu[v_0/\nu]\,]$ *that the given* b *was sampled from* D *conditional on observing* $\alpha_\nu[v_0/\nu]$ *equals* $Pr[v_0 \leftarrow D; v_0 = b]$. ///

Lemmas 1–2 are authentication properties; Lemma 3 is a secrecy property. The IF is parameterized by a function F_{log} and a family of distributions D_{ℓ,P_s,P_r}; each instance of IF is of the form IF$[F_{log}, \{D_{\ell,P_s,P_r}\}]$. The lemmas hold for all values of the parameters.

3 CAIF Devices

We use cryptography to implement IF, eliminating the protected state in atlog and protstore. This cryptographically achieved IF is CAIF. It uses a single fixed, unshared secret.

CAIF is built around two main ingredients: first, the idea of a CAIF *service*, a computational activity with a known *service hash* that serves as its identity (Section 3.1); and, second, two pairs of *instructions* or basic operations to ensure provenance and control access of data passed among services (Section 3.2). Auxiliary operations are also needed to manage services (Section 3.3). Section 3.4 summarizes what being a CAIF device requires.

3.1 CAIF Services

A CAIF device designates some active processes as *services.* A service has an address space such that:

1. Executable addresses are located only within a non-writable *code segment*;
2. A non-shared *heap segment* is readable and writable by this service, but not by any other active process;
3. Other address space segments may be shared with other active processes.

These segments are disjoint, so that code is readable and executable but not writable, while heap is readable and writable but not executable. A program can address them reliably, so that secrets (e.g.) are written into unshared heap rather than shared memory. Moreover:

4. The CAIF device controls when a service is active, and maintains the *hash* of the contents of its *code segment* as its *service identity* or *principal.*

The code segment being immutable, the hash does not change, and CAIF regards it as a non-forgeable identity. To refer to the principal, we often speak simply of its *service hash.*

3.2 CAIF instructions

CAIF offers two pairs of instructions concerning service-to-service information flow. They correspond to iattest and icheck for asserting and checking provenance, and iprotect and iretrieve for escrowing data values and controlling their

propagation. These primitive operations use symmetric cryptography and keys that are derived from a device-local secret unknown to any party called the Intrinsic Secret, combined with one or more service service hashes.

5. An *Intrinsic Secret* within each CAIF device d is shared with no other device or party, and is used exclusively to derive cryptographic keys for the CAIF instructions.

We write *IS* for this intrinsic secret. *IS* is the only secret that the CAIF hardware has to maintain. The hardware design of a CAIF device can help provide assurance that *IS* is not accessible in any way other than key derivation. Implementations may record it as a set of fused wires as in SGX or as a Physically Unclonable Function as in Sanctum [25].

The hardware furnishes four cryptographic primitives at the hardware level, namely a key derivation function kdf_h, a Message Authentication Code mac_h, and an authenticated mode of symmetric encryption enc_h with the decryption dec_h.

A conceptual view of the cryptographic components at the hardware level is in Fig. 1. Triple arrows indicate wires carrying the bits of various secret or

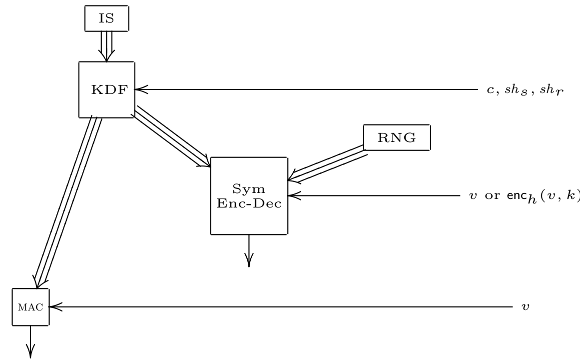

Fig. 1. CAIF Hardware Cryptography Components

random values; it is important that their source and destination are guaranteed by physical connections.

Instructions may *fail* or *succeed*. Failing may be implemented by a transfer of control or terminating the process, or simply by setting a condition code to be checked to determine whether to branch in subsequent instructions.

Provenance via MACs. One pair of primitive instructions uses MACs to identify a service that has generated or endorsed a particular data value v, typically the data occupying a given region of memory defined by a pointer and a length.

`attloc`(v): Computes a MAC on given data v using a key derived from IS and the service hash sh of the current service. If no service is currently executing it fails.

The MAC key k is the result of key derivation via kdf_h:

$$k = \mathsf{kdf}_h(\texttt{"at"},\ IS,\ sh).$$

The instruction computes the MAC $m = \mathsf{mac}_h(k, v)$ of v with k and causes m to be stored. The MAC tag m may subsequently be copied anywhere.

`ckatt`(sh_s, v, m): Checks—given a service hash sh_s of the purported source, data v, and a purported MAC m—whether m is correct. It returns true or false depending whether the purported MAC m equals a recomputed MAC. Thus, letting:

$$k = \mathsf{kdf}_h(\texttt{"at"},\ IS,\ sh_s)$$

the result is true iff $m = \mathsf{mac}_h(k, v)$.

In this way, a service may log itself as the origin or approver of v; any recipient of v or a copy of it, if executing on the same device with the same intrinsic secret IS, can subsequently ascertain its provenance. More specific "intent" on the part of sh_s may be encoded into the content of v.

Thus, `attloc` and `ckatt` provide a device-local mechanism for asserting and confirming provenance.

Protection and provenance via encryption. The second pair of primitive operations uses authenticated symmetric encryption. It produces an encrypted value that can only be decrypted by a stipulated recipient, who successfully decrypts it only if it was prepared by the expected source.

`protfr`(sh_r, v): Computes—given an intended recipient's service hash sh_r and a data value v—an authenticated symmetric encryption of v using a key k derived from IS, the service hash sh of the currently executing service, and sh_r. If no service is currently executing it fails.

The encryption key k is computed via key derivation function kdf_h as:

$$k = \mathsf{kdf}_h(\texttt{"pf"},\ IS,\ sh,\ sh_r).$$

The third component is the service hash sh of the currently running service, and sh_r is its intended recipient. The instruction computes the encrypted value:

$$e = \mathsf{enc}_h(v, k)$$

which is stored back into a suitable region of memory. The resulting e may subsequently be copied anywhere.

`retrfm`(sh_s, e): Decrypts e—given an expected source's service hash sh_s and a purported encryption e—using key k derived from IS, the source's service hash sh_s, and the code hash sh of the currently executing service. Fails if the decryption fails, or if no service sh is executing.

The decryption key k is computed via kdf_h as:

$$k = \mathsf{kdf}_h(\texttt{"pf"},\ IS,\ sh_s,\ sh).$$

The service hash sh of the currently executing service is now the last component, and the source sh_s is the previous one. If $e = \mathsf{enc}_h(v, k)$, the plaintext v is stored back, typically into unshared heap.

These instructions ensure provenance with access control, since only the intended recipient can check the source, and only device-locally. They also ensure confidentiality, since in the absence of a `retrfm` by the intended recipient, observers cannot distinguish whether a data value v or another v' of the same length was protected.

3.3 Auxiliary operations

We also need some auxiliary operations on services:

`create-service`: Creates a service with the code contained in a buffer of memory, plus some other resources including a newly allocated unshared heap. The service hash is ascertained to be used by the CAIF mechanism.
The resulting service does not execute immediately, but is placed on a list of runnable services.

`start-service`: Given a runnable service, start it executing with access to the values of any additional parameters.

`yield`: Stop executing the current service, retaining its state for future `start-service`s.

`exit-service`: Zero the unshared heap of the service and eliminate it from the list of runnable services.

A service may be created from any executable, even if they do not use the CAIF instructions; whatever executable it is will, however, be indelibly associated with it through the service hash. Thus, its identity is always correctly reflected if it uses the four core CAIF instructions. Varying implementations can use different versions of the auxiliary instructions.

3.4 CAIF devices

By a *CAIF device* we mean a hardware device that enables the CAIF instructions in Section 3.2 to be performed by services of the kind given in Section 3.1; the auxiliary operations in Section 3.3 may be carried out using a combination of software and hardware.

Definition 2. *A* CAIF state *is an intrinsic secret IS. No* CAIF *command modifies the* CAIF *state.*

CAIF behaviors $\langle (c_j, p_j, r_j) \rangle_{j<i}$ *are behaviors containing* CAIF *commands* c_j*, active principals—i.e. code hashes—*p_j *and command results* r_j*.*

Cryptographic choices. CAIF devices should be equipped with strong cryptographic primitives. In particular, implementors should endeavor to ensure:

Key derivation: The key derivation function kdf_h is indistinguishable from a random function. We measure it by the *pseudorandomness advantage* $\mathsf{Adv}_{prf}(A, k, q)$ of any adversary algorithm A aiming to separate kdf_h from a random function using up to q queries to kdf_h and a polynomial number of operations relative to the security parameter k. See [16].

MAC: The `attloc` and `ckatt` primitives use a Message Authentication Code mac_h based on a hash function with the collision resistance property. The code hashing algorithm is also collision resistant. We measure this by its *existential unforgeability advantage* $\mathsf{Adv}_{eu\text{-}mac}(A, k, q)$ for A aiming to generate a correct MAC or code hash for a bitstring that was not queried.

Encryption: The `protfr` and `retrfm` primitives use an enc_h secure against chosen ciphertext attacks [31]. We measure it by two values. The *chosen ciphertext indistinguishability advantage* $\mathsf{Adv}_{indCCA2}(A, k, q)$ measures A's ability to distinguish encryptions of two different plaintexts, and the *ciphertext unforgeability advantage* $\mathsf{Adv}_{eu\text{-}enc}(A, k, q)$ measures A's ability to generate a value that will decrypt under an unknown key.

We are in effect recommending a family of CAIF devices, parameterized by a security parameter k. Section 4 argues that as k increases and the advantages just described decrease, the CAIF devices become much harder to distinguish from the ideal functionality.

Hardware implementation. A hardware implementation of CAIF is under development, with FPGAs for convenience. ASICs will subsequently be needed, e.g. to protect *IS* properly.

The CAIF "special instructions" are implemented not as instructions, but with stores to a memory-mapped peripheral region of the FPGA followed by loads from it. FIFOs ensure that the service's view of the process is atomic, i.e. that none of the ciphertext for `protfr` can be observed until all of the plaintext has been committed.

An open-source RISC-V soft core provides the instruction set for the processor functionality. Keystone [26] suggests a provisional way to enforce the memory protection in Section 3.1, items 1–3. A more complete CAIF implementation will provide memory protection assurance directly in hardware, using a simple layout of services in physical memory.

4 CAIF securely implements IF

We next prove that a CAIF device is close in behavior to the ideal functionality IF, where *close* is quantified by the cryptographic properties of the primitives kdf_h, mac_h, $(\mathsf{enc}_h, \mathsf{dec}_h)$ used in the construction.

Oracles. A computational process $A^{\mathcal{F}}$ may make queries to a state-based process $\mathcal{F}$, receiving responses from it. We refer to the latter as an *oracle*.

In any *state* any *query* determines a distribution $\mathcal{D}$ on *(next-state,result)* pairs. A *behavior* or *history* is a sequence of *query-result* pairs, such that there is a sequence of states where the first state is an *initial state* and, for each successive state s, that state and the result are in the $\mathcal{D}$-support for the previous state and the current query. More formally:

Definition 3. *An* oracle *is a tuple* $\mathcal{O} = \langle \Sigma, Q, I, R, \delta \rangle$ *such that* $I \subseteq \Sigma$, $\bot \in R$, *and* $\delta \colon \Sigma \times Q \to \mathcal{D}(\Sigma \times R)$.

Σ is the set of states, I being the initial subset. Q is the space of possible queries. The function δ is the probabilistic transition relation. An alternating finite sequence

$$\langle (\sigma_0, \bot), q_0, (\sigma_1, r_1), q_1, \ldots, q_i(\sigma_{i+1}, r_{i+1}) \rangle \tag{1}$$

is a trace *of $\mathcal{O}$ iff $\sigma_0 \in I$, and for all $j < i$,*

1. *$q_j \in Q$, $\sigma_{j+1} \in \Sigma$, and $r_{j+1} \in R$; and*
2. *$(\sigma_{j+1}, r_{j+1}) \in \mathsf{supp}(\delta(\sigma_j, q_j))$.*

The probability of any trace of $\mathcal{O}$ is determined by the Markov chain condition, i.e. the product of the (non-zero) probabilities of (σ_{j+1}, r_{j+1}) in $\delta(\sigma_j, q_j)$.

A behavior *or* history *of $\mathcal{O}$ is a finite sequence of pairs $\langle\, (q_j, r_{j+1}) \,\rangle_{j<i}$ such that for some sequence $\langle \sigma_j \rangle_{j \le i}$, the sequence (1) is a trace of $\mathcal{O}$.* ///

We use *behavior* and *history* interchangeably.

$A^{\mathcal{F}(s_o,\cdot)}$ denotes running of A with oracle access to $\mathcal{F}$ with initial state s_0. We write $A^{\mathcal{F}(\cdot)}$ if the initial state is clear. $\mathcal{F}(\cdot)$ is function-like, returning results r_j for queries (c_j, P_j).

A partial run of $A^{\mathcal{F}(\cdot)}$ induces a history $\mathcal{H}$ of $\mathcal{F}(\cdot)$. If it completes with answer a, having induced the $\mathcal{F}(\cdot)$-history $\mathcal{H}$, then we write $(a, \mathcal{H}) \twoheadleftarrow A^{\mathcal{F}(\cdot)}$. When we do not need $\mathcal{H}$, we write $a \leftarrow A^{\mathcal{F}(\cdot)}$ as usual.

Where $\mathcal{H} = \langle (q_i, r_i) \rangle_i$ is a history, $\mathsf{query}(\mathcal{H})$ refers to the sequence $\langle q_i \rangle_i$. We write $v \in_i \mathcal{S}$ when $\mathcal{S}$ is a sequence and v occurs in the ith position of $\mathcal{S}$.

4.1 Defining the CAIF properties

A CAIF device is an oracle with unchanging state, the intrinsic secret IS, with constant length $|IS|$. Its commands are the four instructions of Section 3.2, syntactic objects consisting of an instruction name together with values for the arguments. The result $\bot$ signals instruction failure.

The CAIF functionalities form a family $\{C\}_k$ of oracles depending on cryptographic primitives they use; these depend on a security parameter k for key sizes, block sizes, etc.

An instance of the IF is also an oracle. Its state consists of atlog and $\mathsf{protstore}$. The transition relation δ depends on F_{log} and the family of distributions D_{ℓ,P_s,P_r}, since these determine the resulting state σ_{j+1} and the result r_{j+1}. We identify the instruction `attloc` with $\mathsf{iattest}$, and so forth.

Our CAIF and IF oracles take queries (c, p), where c is a syntactic command and p is a service principal (i.e. a service hash). As in Section 2, we write behaviors in the form $\langle\, (c_j, P_j, r_j)\,\rangle_{j<i}$ rather than $\langle\, ((c_j, P_j), r_{j+1})\,\rangle_{j<i}$.

$A_{q,t}$ denotes the adversary A instrumented to halt immediately if it exceeds q oracle queries or t computational steps. Adversaries are possibly stateful, and may operate in phases; $A_{q,t}$ causes all phases of A combined make at most q queries and at most t computational steps before halting.

Definition 4 (caif properties). *Let $\{C\}_k$ be a CAIF functionality and k a security parameter for which C_k is defined.*

1. *The* attestation unforgeability advantage *of A is:*

$$\begin{aligned}&\mathsf{Adv}_{a\text{-}u}(A, C, k, q, t)\\ &\quad = \Pr[((p, v, \tau); \mathcal{H}) \twoheadleftarrow A_{q,t}^{C_k(\cdot)} :\\ &\qquad \exists i\,.\, \forall j < i\,.\, (\mathsf{icheck}(p, v, \tau), p', \mathsf{true}) \in_i \mathcal{H} \wedge\\ &\qquad\quad (\mathsf{iattest}(v), p, \tau) \notin_j \mathcal{H}]\end{aligned}$$

2. *The* protection unforgeability advantage *of A is:*

$$\begin{aligned}&\mathsf{Adv}_{p\text{-}u}(A, C, k, q, t)\\ &\quad = \Pr[((p, v, \eta); \mathcal{H}) \twoheadleftarrow A_{q,t}^{C_k(\cdot)} :\\ &\qquad \exists i\,.\, \forall j < i\,.\, (\mathsf{iretrieve}(p, \eta), p', v) \in_i \mathcal{H} \wedge\\ &\qquad\quad (\mathsf{iprotect}(v, p'), p, \eta) \notin_j \mathcal{H}]\end{aligned}$$

3. *Let $I^{(\cdot)}(m, P_s, P_r)$ query $(\mathsf{iprotect}(P_r m), P_s)$ with result η. The* protection confidentiality advantage *of A is $\mathsf{Adv}_{p\text{-}c}(A, C, k, q, t) = |p_0 - p_1|$ where $p_b =$*

$$\begin{aligned}\Pr[&(m_0, m_1, P_s, P_r, \alpha) \leftarrow A_{q,t}^{C_k(\cdot)}(\bot);\\ &\eta \leftarrow I^{C_k}(m_b, P_s, P_r);\\ &(x; \mathcal{H}) \twoheadleftarrow A_{q,t}^{C_k(\cdot)}(\alpha, \eta) :\\ &x = 1 \wedge |m_0| = |m_1| \wedge\\ &(\mathsf{iretrieve}(P_s, \eta), P_r) \notin \mathsf{query}(\mathcal{H})]\end{aligned}$$

4.2 Proving the properties

Now we can state claims defined in terms of advantages. Let k be a security parameter for which CAIF is defined.

Lemma 4 (Attestation unforgeability). *There are reductions $\Gamma_{1,q}$ and Γ_2 with additive computational overhead, such that for any A, $\mathsf{Adv}_{a\text{-}u}(A, \textsc{caif}, k, q, t) \leq$*

$$\begin{aligned}&q\mathsf{Adv}_{eu\text{-}mac}(\Gamma_{1,q}(A), k, q) +\\ &\mathsf{Adv}_{prf}(\Gamma_2(A), k, q+1) \qquad ///\end{aligned}$$

$$\overbrace{\underbrace{\Pr[A^{\mathcal{O}_0(\cdot)} = 1] \qquad \Pr[A^{\mathcal{O}_1(\cdot)}}_{q^2\mathsf{Adv}_{indCCA2}(\Gamma_{7,q}(A))} \underbrace{= 1] \qquad \Pr[A^{\mathcal{O}_2(\cdot)}}_{\mathsf{Adv}_{prf}(\Gamma_8(A))} \underbrace{= 1] \qquad \Pr[A^{\mathcal{O}_3(\cdot)} = 1]}_{q[\mathsf{Adv}_{a\text{-}u}(\Gamma_{9,q}(A)) + \mathsf{Adv}_{p\text{-}u}(\Gamma_{10,q}(A))]}}^{\mathsf{Adv}_{ind}(A, \text{CAIF}, k, q)}$$

Fig. 2. Implementation theorem proof strategy

Proof: See [16], , as for the remaining proofs.

Lemma 5 (Protection unforgeability). *There are reductions $\Gamma_{3,q}$ and Γ_4 with additive computational overhead $t_3(k,q)$ and $t_4(k,q)$ respectively, such that for any A, $\mathsf{Adv}_{p\text{-}u}(A, \text{CAIF}, k, q, t) \leq$*

$$q\mathsf{Adv}_{eu\text{-}enc}(\Gamma_{3,q}(A), \mathcal{E}, k, q, t + t_3(k,q)) + \mathsf{Adv}_{prf}(\Gamma_4(A), \mathsf{kdf}_h, k, q+1, t + t_4(k,q)) \qquad ///$$

4.3 Proving implementation of the ideal functionality

The properties above help us state and prove a powerful claim: CAIF provides a secure implementation of the ideal functionality IF discussed in Section 2.

Let IF be the instance $\text{IF}[F_{log}, \{D_{\ell,P_s,P_r}\}]$ where, for a CAIF device with given crypto primitives and a particular intrinsic secret IS:

$F_{log}(v, P) = \mathsf{mac}_h(sk, v)$
where $sk = \mathsf{kdf}_h(\texttt{"at"}, IS, P)$;
D_{ℓ,P_s,P_r} is the distribution generated by $\mathsf{enc}_h(0^\ell, sk)$
where $sk = \mathsf{kdf}_h(\texttt{"pf"}, IS, P_s, P_r)$.

Theorem 1. *Let $\{C_k\}$ be a CAIF functionality and let k be a security parameter for which which C is defined. Let $\mathsf{Adv}_{imp}(A, C, k, q, t)$ be defined to be*

$$|\Pr[A^{C_k(\cdot)} = 1] - \Pr[A^{\text{IF}_k(\cdot)} = 1]|.$$

There are reductions $\Gamma_{7,q}$, Γ_8, $\Gamma_{9,q}$, and $\Gamma_{10,q}$ with computational overhead times $t_7(k,q)$, $t_8(k,q), t_9(k,q)$, and $t_{10}(k,q)$ respectively, together with the reductions $\Gamma_{1,q}$, Γ_2, $\Gamma_{3,q}$, and Γ_4 of Lemmas 4–5, s.t. for all $q < W$,

$$\begin{aligned}\mathsf{Adv}_{imp}(A, \text{CAIF}, k, q, t) \leq& \\ & q^2\mathsf{Adv}_{indCCA2}(\Gamma_{7,q}(A), k, q) + \\ & \mathsf{Adv}_{prf}(\Gamma_{10}(A), k, q) \\ & q^2(\mathsf{Adv}_{eu\text{-}mac}(\Gamma_{1,q}(\Gamma_{9,q}(A)), k, q)) + \\ & q(\mathsf{Adv}_{prf}(\Gamma_2(\Gamma_{9,q}(A)), k, q+1)) \\ & q^2(\mathsf{Adv}_{eu\text{-}enc}(\Gamma_{3,q}(\Gamma_{10,q}(A)), k, q)) + \\ & q(\mathsf{Adv}_{prf}(\Gamma_4(\Gamma_{10,q}(A)), k, q+1))\end{aligned}$$

Proof sketch: The proof operates as a hybrid argument regarding four probabilities, namely the probabilities that $A^{\mathcal{O}_i(\cdot)}$ outputs 1 for various oracles $\mathcal{O}_i$ for $0 \leq i \leq 3$. The oracles are defined as follows:

$\mathcal{O}_0 = \text{IF}$.
$\mathcal{O}_1$ implements IF, except that for iprotect queries, we select the candidate values c by encrypting the input value v rather than $0^{|v|}$.
$\mathcal{O}_2$ implements $\mathcal{O}_1$ except that it uses kdf_h with an intrinsic secret IS instead of using R_k.
$\mathcal{O}_3 = C_k$.

We construct a reduction for each gap, summarized in Fig. 2. Lemmas 4–5 bound the rightmost term in Fig. 2. ///

Hence, if CAIF uses strong cryptography:

Corollary 1 *If a* CAIF *device has pseudorandom* kdf_h*, collision-resistant* mac_h *and code hashing, and IND-CCA2* enc_h*, then that* CAIF *device is computationally indistinguishable from an instance of* IF. ///

5 Assured Remote Execution Strategy

In the remainder, we aim to show that the CAIF mechanism can achieve assured remote execution, even against a strong, code-selecting adversary, in a verifiable way.

5.1 Achieving Assured Remote Execution

We build up our guarantees of assured remote execution in a succession of steps, and offer two versions. The main version does not depend on any digital signature algorithm remaining secure throughout a device's lifetime. It establishes a shared secret k_s used to provide trust in new signature algorithms when they are developed and deployed to the device, or whenever larger key sizes needed for the same algorithm.

A simpler version that does not use a shared secret is for short-lived devices that will never outlast validity of the signature algorithm and key size favored at initialization. It effectively starts near the end of the first strategy.

Each version needs to start off with an assumption. Our device must be started running a known program at least once, at the beginning of its lifetime. The manufacturer—already trusted to produce the hardware correctly—initializes the underlying hardware with a compliant service to run first. This *anchor* service must run *only* in a secure environment. Devices undergo a state change, such as a switch or a wire that can be fused, to prevent re-execution after anchoring.

The device itself has a permanent uniquely identifying number, its *immutable identifier imid*. We use it as a name for the device; *imid*s, like other names, may be publicly known.

The peer in anchoring is the controlling authority, the *device authority* DA. The DA comes to share a secret k_s in step 1. The DA must store it securely and make it available later.

Starting with our assumption, the first anchoring version is:

0. We assume *local* execution of a single service, the *anchor service*, specifically at the time of initialization, in a context suitable for secure initialization.
1. The DA and the anchor service establish a shared secret k_s. The anchor service protects k_s for the exclusive use of a particular recipient service svc_1. It then zeros its memory and exits.
   A state change prevents rerunning the anchor service.
2. We choose svc_1 to be a service that receives authenticated requests from DA; each request specifies a service hash *sh* for which to prepare a derived shared secret. It is a *symmetric key distributor service*. It protects a symmetric key $\mathsf{kdf}(k_s, sh)$, for the exclusive use of the service with hash *sh*. It then zeros its memory and exits.

To escape from using shared secrets to infer assured remote execution, we establish a *signing key delegation service*. Whether near the time of initialization or long afterwards, possibly repeatedly, we install new programs and authorize them using the symmetric key distributor:

3. A *set-up service* with hash *suh* generates a signature key pair $(\mathsf{dk}, \mathsf{dvk})$. It interacts with a certifying authority CA operated by DA using the shared secret $\mathsf{kdf}(k_s, suh)$, providing a proof that it holds the signing key dk, and obtaining a certificate that associates the verification key dvk with a service hash *dsh*, the device's *imid*, and some supplemental values.
   The set-up service protects dk for the exclusive use of the service *dsh*. It then zeros its memory, and exits.
4. The service with hash *dsh*, when invoked with a target service hash *sh*, generates a new signing key pair $(\mathsf{sk}, \mathsf{vk})$. It emits a certificate-like binding $[\![\ldots imid, sh, \mathsf{vk}, \ldots]\!]_{\mathsf{dk}}$ signed with dk. It protects sk for the exclusive use of the service *sh*. It then zeros its memory, and exits.

Each step builds additional assured remote execution power, starting from 0 that simply assumes a local execution in a specific metal room environment. Step 1 assures remote execution for a *single* service svc_1, with evidence available only to DA holding k_s. Step 2 allows *any* service *sh* to receive a secret, but evidence of *sh*'s remote execution is available only to the DA or those sharing the secret. With step 4, the progression is complete: *any* service *sh* can receive a signing key to document its remote execution, and any principal willing to trust the CA's certificate can evaluate the evidence. No shared secrets are active as *sh* executes, providing general assured remote execution.

If the device will never outlive a signing algorithm and keysize, a simpler protocol is enough. We use the set-up service of step 3 as the anchor service. It obtains a certificate on dvk from the DA, allowing the signing key delegation service of step 4 to provide signing keys to any service *sh*. Shared secrets are not needed; however, if algorithm or key used are superseded, assured remote execution cannot be reestablished. We will not discuss this version further.

5.2 Compliant roles and adversary roles

Over time, a device has many active processes that transmit and receive messages and use the CAIF instructions. Many of these will be unpredictable, either because the adversary chose them, or because they were simply poorly written. They may use the CAIF instructions with any values they can obtain; CAIF just ensures they do so under their own service hash. They are wildcats. We call them *adversarial services*.

However, some programs, having been carefully constructed for specific behaviors such as the ones we have just described, may have no other relevant behaviors. They do not use these keys for any other messages, nor use the CAIF instructions in any other relevant way. We call them *compliant services*.

A compliant service complies with one or more *role specifications*, and never performs relevant actions except as the role dictates. The predicate compliant is true of service hashes of programs, intuitively, that we have vetted and judge compliant. If a service hash *sh* is compliant, the consequence is:

- every use of the CAIF instructions with active service hash *sh* belongs to an instance of a specified compliant role,

i.e. the roles with behaviors defined in Sections 6–7. The compliant property always arises as an *assumption* in analysis; we will not present methods here for proving services are compliant, presumably a task for program verification.

Adversary activities using the CAIF instructions are *wildcats*; they use service hashes not assumed compliant.

Wildcat roles. We specify the adversary's powers—beyond the usual network adversary's powers to interrupt, redirect, and synthesize messages, to extract and retain their contents, and to execute cryptographic operations using any keys they may possess—by three *wildcat* roles with names beginning `wc-`, so-called because they may use the IF instructions in whatever unexpected patterns would benefit the adversary:

wc-protect causes an iprotect instruction with current service sh_s, recipient sh_r, and value v.

wc-retrieve causes an iretrieve instruction and transmits v for adversary use.

wc-attest causes an iattest instruction, logging v with the current service sh_s in the atlog.

A wildcat role for `ckatt` is unnecessary in this context, because it is a conditional that produces no new data. The cases are represented for protocol analysis via pattern matching.

Rules on the wildcat roles. The wildcat roles obey rules saying that if *sh* is the active hash in a wildcat role, then *sh* is not compliant. So CAIF instructions with compliant service hashes occur *only* in rule-bound, non-wildcat roles.

This is a strong adversary model: the adversary may install any programs and execute them as services, using `protfr` etc. as desired. However, if a program has

the same service hash we assume compliant, then it will have the same behavior we specified in a compliant, non-wildcat role. The adversary can also run our services, unless special provisions prevent some from running, as in anchoring (step 1).

6 Anchoring a CAIF Device

When a CAIF device has been manufactured and arrives at the warehouse of the purchaser—or alternatively, just before it is shipped—it can be *anchored*.

Anchoring runs a known program on the fresh CAIF device in a secure environment. This may require shielded cables to the device, or wireless communications shielded in a Faraday cage. Thus, we will call this the *ceremony in the metal room*.

The metal room provides *authenticated* and *confidential* channels between DA and the device.

6.1 Anchoring with Shared Secrets

Anchoring delivers a shared secret k_s. Subsequently the device and DA use k_s for symmetric encryptions and MACs.

The anchor service. The steps of anchoring are:

1. DA turns on d and starts d running the *symmetric anchor service*, with service hash $anch$.
2. DA observes the device identifier $imid$, and prepares a secret seed r and a nonce n. DA transmits

$$imid, anch, dstrh, n, r,$$

   $dstrh$ being the service hash to obtain the secret k_s.
3. The service $anch$ checks its device has id $imid$, and its service hash is $anch$. It then computes

$$k_s = \mathsf{kdf}_h(r, imid),$$

   and replies with n to confirm completion.
4. It uses `protfr` to protect k_s and its identity $anch$ for service $dstrh$ exclusively; it zeroes its memory and exits.

The effect is to make k_s available on $imid$ only to the service with service hash $dstrh$. The seed r may itself be derived from a group seed r_0 by differentiating it for $imid$, e.g. using $r = \mathsf{kdf}_h(r_0, imid)$. This eases managing long-term secrets.

Since the anchor service $anch$ must run *only* with a secure channel to its peer, we need an irreversible state change after which $anch$ will no longer accept messages.

The ceremony in the metal room is done, but k_s remains permanently available to $dstrh$ to secure remote interactions.

6.2 Analyzing symmetric anchoring

We now analyze the ceremony in the metal room.

A role on the device represents the *anchor* service, together with a role *dev-init-imid* to initialize a device's immutable ID *imid* to a fresh value. A pair of

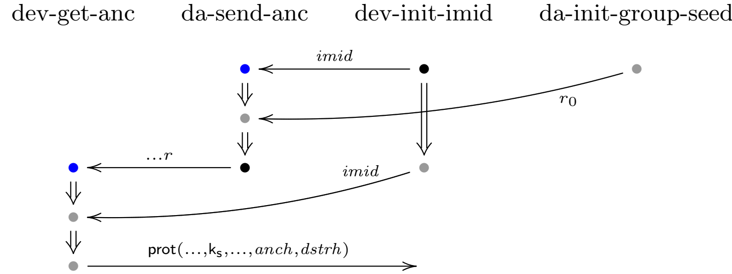

Fig. 3. Metal room activity, symmetric anchoring

roles represent the *DA*'s interaction with the device. One does set-up to manage the shared secrets, creating a *group seed*. The shared secrets are derived from it by hashing with *imid*s. A second *sends the anchor* secret during the ceremony in the metal room.

Our analyses all assume that the anchor service hash *anch* is compliant, i.e. that the adversary cannot perform wildcat actions under hash *anch*. We ascertain that k_s and various derivatives of it remain unavailable to the adversary.

A significant verification at the metal room layer determines what happened if the device *anchor* role runs to completion. We assume the metal room ensures both authenticity and confidentiality. We find that the anchor secret k_s is obtained from the *DA* sending the secret r. The two parties obtain the device's *imid* from the same device events. And the *DA* has properly derived the anchor secret from its group seed (Fig. 3).

In Fig. 3, vertical columns represent successive actions—reading downwards along the double arrows—of an instance of the role named at the top; single arrows represent propagation of messages or of values in device-local state (`protfr` records, secure seed storage at the *DA*). Message transmission and reception are shown as black nodes • and blue nodes •, resp. Local state reads and writes are gray nodes •. More comments on the diagrams are at the end of Section 7.4.

In subsequent steps, as we add more roles to model subsequent activities we reverify these properties, since protocol interactions could undermine them.

6.3 Trust chains

We keep track of "chains of custody" for trust, meaning the sequence of services—starting from an anchor service—that obtained previous keys and generated new

keys to protect for their successors. We store trust chains with the keys that they validate in `protfr` records. We also deliver trust chains in messages between parties. The parties check these agree to trace the trustworthiness back to the anchoring program.

Trust chains are effectively lists of service hashes. If *trch* is a trust chain, we represent the effect of pushing a new hash h to it as $h :: trch$, using a *cons* "::" operation. We also write $\langle h_0, h_1, \ldots, h_k \rangle$, e.g. in which h_k is the oldest entry.

We have successfully designed and verified protocols relying on trust chains of length up to five (see Section 7).

6.4 The symmetric key distributor service

The anchor service may be used with any *dstrh*. A useful *dstrh* is a *symmetric key distributor service.* The symmetric key distributor program retrieves k_s and uses it to decrypt a message from the *DA*. This message contains:

- a target service hash *acth* for which a new symmetric key $\mathsf{kdf}_h(k_s, acth)$ should be derived;
- a trust chain $trch = \langle dstrh, anch \rangle$;
- a message *payld* to pass to the service *acth* when it runs.

The distributor service checks the trust chain, i.e. that its service hash is *dstrh* and it retrieved its k_s from the source service *anch*. If so, *dstrh* derives a key $k = \mathsf{kdf}_h(k_s, acth)$. It protects the trust chain, the payload, and this key k for *acth*:

$$\texttt{protfr}\ acth\ (imid,\ (acth :: trch),\ payld,\ k)$$

after which it zeroes its memory and exits. As a consequence, the *DA* now knows a secret, namely $k = \mathsf{kdf}_h(k_s, acth)$, which on device *imid* can only by obtained by the service *acth*.

This k enables the *DA* to create an authenticated, confidential channel to *imid* where only service *acth* can be the peer.

6.5 Analyzing symmetric key distribution

This layer of analysis has a role representing the *symmetric key distributor* service on the device, together with a role *use-it* that retrieves the distributor's key and uses it generate a confirmation message. A third role on the *DA* delivers the *distribution request* to the device. It terminates successfully after receiving a confirmation message from *acth*.

The analysis for compliant code hashes for *anch*, *dstrh*, and the target service *acth* appears in Fig. 4. The distribution service will be uses k_s generated by the behavior in Fig. 3; it handles the request and derives k, protecting it for *acth*. The latter uses k to prepare a confirmation for the requester. The value *seed* is stored in state in the *DA*, while the value k_s is stored in the device state inside a *protect-for* record, generated in the lower left node of Fig. 3.

Fig. 4. Distributor request

If *acth* is not assumed compliant, a separate analysis shows that, as an alternative to Fig. 4, `wc-retrieve` can expose k. This is correct in our strong adversary model.

Because the arrows in Fig. 4 point out of the box at the upper right, the behaviors in the remainder of the diagram cannot come before the anchoring ceremony. But because no arrows point *into* the box at the upper right, the remainder may come as long as desired after the anchoring ceremony. So the distributor may be used to set up keys for software long after the ceremony in the metal room, e.g. when the device is in orbit on a satellite. This functionality assures long-term remote execution without local contact.

7 Delegating Signing Keys

A *delegation service* holding a signing key dk behaves as follows to generate a signing key for a service *sh*:

1. It generates a fresh target signature key pair (sk, vk);
2. It binds the new verification key vk to *sh* and the device *imid* and additional information by emitting a certificate-like message signed with dk. We call this a *delegation certificate*.
3. It protects the signing key sk and additional information for the exclusive use of service *sh*. It then zeroes its unshared memory and exits.

A recipient who knows the delegation service generated the delegation certificate can thus infer that *sh* has been active on *imid* if a message signed with sk is observed. To ensure it generates these certificate, we use a *delegation set-up service* to obtain a CA certificate for dk. Suppose that the delegation service hash is *dsh* and the set-up service's hash is *suh*.

The delegation set-up service acts as follows:

1. It obtains a *certify request* ultimately from the CA containing a fresh *serial* number for the resulting certificate;
2. It generates a signing key pair (dk, dvk) for *dsh*;

3. It transmits a proof-of-possession message using the signing key dk to the CA on an authenticated channel;
4. It receives a certificate; and
5. It protects dk together with additional information for the sole use of *dsh*; it zeros its memory and exits.

The additional information of the set-up and delegation services includes a trust chain asserting full data provenance (Section 6.3).

7.1 Assumptions and security goal for delegation

Our delegation scheme relies on the assumptions:

(i) the CA is uncompromised;
(ii) the CA interacts with *suh* on *imid* through an authenticated channel when certifying dvk.

The anchoring in Section 6 provides ways to establish the authenticated channel, with a specific service as endpoint, required in Assumption (ii). Moreover, for a particular target service with service hash *sh* one may assume:

(iii) the service *sh* uses sk for signing messages, but not in any other way; hence *sh* does not disclose sk.

Suppose we observe: A digital certificate from CA binding dvk to *dsh* on *imid*; a delegation certificate binding vk with *sh* on *imid*; and a message $m = [\![m_0]\!]_{\mathsf{sk}}$ verifying under vk.

Our *main security goal* is: When assumptions (i) and (ii) hold, and (iii) holds for this *sh*, then, on device *imid*:

1. The delegation set-up service has generated dk, obtained the certificate on dvk and *dsh*, and protected dk solely for the delegation service *dsh*;
2. The delegation service *dsh* generated $(\mathsf{sk}, \mathsf{vk})$, emitted the certificate on *sh* and vk, and protected sk for the sole use of the service *sh*;
3. This service *sh* used sk to sign m_0, yielding m.

The observer thus infers, subject to (i)–(iii), that the delegation process proceeded correctly, and that *sh* is responsible for m_0. This is the assured remote execution claim for m_0.

7.2 Message forms

We use tags to distinguish tuples of components that might be confused. Here, we only show the components, not the tags. The *certification request* contains the components:

$$imid, dsh, suh, trch, serial, CA,$$

where $trch$ is a trust chain acceptable to the CA. The proof-of-possession is signed with the signing part of the key pair and contains the verification key:

$$[\![\, serial, imid, dsh, suh, trch, \mathsf{dvk} \,]\!]_{\mathsf{dk}};$$

the CA must ascertain that it arrives on an authenticated channel from service suh running on $imid$. The anchoring enables this. The resulting certificate is of the form:

$$[\![\, imid, dsh, suh, trch, serial, \mathsf{dvk} \,]\!]_{CA}.$$

The set-up service protects $(\mathsf{dk}, \mathsf{dvk})$ together with an expanded trust chain $trch' = dsh :: suh :: trch$. The delegation service then retrieves $trch'$ with the key pair $(\mathsf{dk}, \mathsf{dvk})$. When generating a key pair $(\mathsf{sk}, \mathsf{vk})$ for a target service sh, it protects $sh :: trch'$ with $(\mathsf{sk}, \mathsf{vk})$ for sh, and emits a delegation certificate

$$[\![\, imid, sh, trch', n, \mathsf{vk} \,]\!]_{\mathsf{dk}}.$$

7.3 Delegation analysis

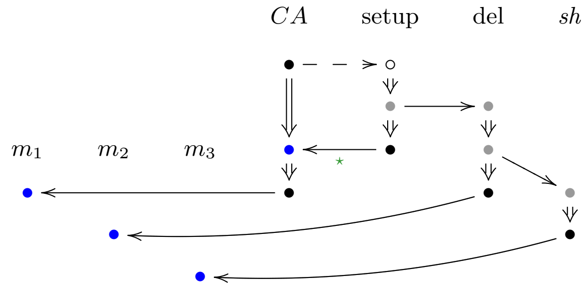

Fig. 5. Message m_3 signed with delegated key; $\star$ authenticated channel

We analyze the delegation mechanism over a generic authenticated channel by querying what must have occurred if the following three messages are observed:

m_1: a certificate $[\![\, imid, dsh, suh, trch, serial, \mathsf{dvk} \,]\!]_{CA}$ for the delegation verification key dvk;

m_2: a delegation certificate $[\![\, imid, sh, trch', n, \mathsf{vk} \,]\!]_{\mathsf{dk}}$ for the service sh's key vk; and

m_3: a message $[\![\, m_0 \,]\!]_{\mathsf{sk}}$ in which m_0 is signed by some sk forming a key pair $(\mathsf{sk}, \mathsf{vk})$ with the certified vk.

These messages are in the three left columns of Fig. 5.

Fig. 5 shows the single result of CPSA's analysis under the assumptions (i)–(iii) from Section 7.1. CPSA determines this is the only way that the certificates and signed message $[\![\, m_0 \,]\!]_{\mathsf{sk}}$ can be observed, subject to (i)–(iii).

In this scenario, $[\![m_0]\!]_{\mathsf{sk}}$ was in fact generated by the expected (rightmost) role instance *sh*, which obtained its key from the delegation service to its left; that in turn generated the certificate on vk and obtained its signing key dk from the delegation set-up service preceding it, which in fact interacted with the CA to generate the certificate.

Assured remote execution. Fig. 5 shows the assured remote execution guarantee, subject to assumptions (i)–(iii). Messages signed by sk come from the program *sh* on device *imid*.

As usual, we can build authenticated and confidential channels to *sh* on top of this, e.g. using m_0 to send a public key for a Key Encapsulation Mechanism [42].

7.4 Adapting delegation to the anchor

Anchoring allows the symmetric key distributor (Section 6.4) to generate $k_{sud} = \mathsf{kdf}_h(k_s, suh)$; *suh* can use it to encrypt traffic to the CA, namely the proof-of-possession.

The symmetric key distributor receives a payload *payld* from the *DA* when deriving a key, which it protects for the recipient with the key. We use the CA's certify request $imid, dsh, suh, trch, serial, CA$ as this *payld*. The delegation set-up service retrieves this using `retrfm`, together with a trust chain $trch_1$ and the derived key k_{sud}. The trust chain *trch* is the *DA*'s acceptable chain of custody for k_{sud}, while $trch_1$ contains its actual history, as analysis confirms. The set-up service does not proceed unless $trch = trch_1$.

Since *anch*, *dstrh*, *suh*, and *dsh* are all in *trch* and thus in the certificates, we assume they are all compliant. The CA emits a certificate only when the trust chain is acceptable.

If the target service hash *sh* is also compliant, analysis yields a single possibility shown in Fig. 6 that enriches the Fig. 5 as expected. The proof-of-possession *pop* is encrypted under k_{sud} on the $\star\star$ arrow from *suh* to CA.

If *sh* may not be compliant, there is an additional possibility in which a wildcat retrieve role extracts and discloses sk, instead of *sh*'s run, as expected.

About the diagrams. We have redrawn and simplified CPSA's diagrams. We reordered role instances for clarity. We grouped copies of Fig. 3 in a single box. We combined adjacent state nodes that are distinct for CPSA. CPSA also emits information about messages transmitted and received, etc., mentioned only in accompanying text.

8 Related Work

8.1 Trusted Execution Environments

CAIF is a kind of TEE, like Intel's SGX and TDX [13,23], AMD's SEV [24], and research such as Sancus [33,34] and Sanctum [14] and Keystone [26] for RISC-V. Schneider et al.'s recent survey on TEEs and their implementation choices [41]

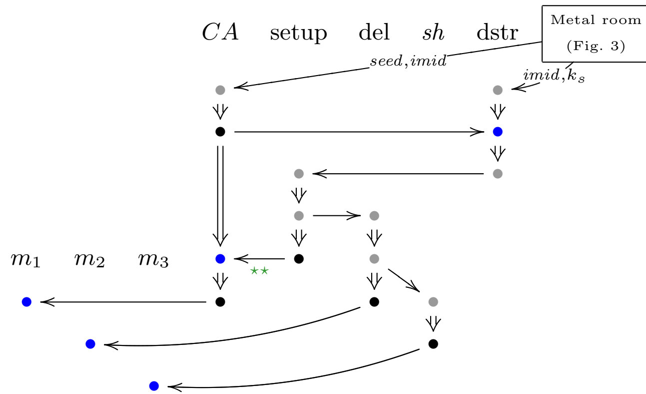

Fig. 6. Message m_3 signed, with anchoring and ⋆⋆ encrypted

identifies four main security properties for TEEs (p. 1). One is is a weak, launch-time version of assured remote execution; the second covers our address space requirements (Section 3.1, 1–2); the third, concerning trusted IO, lies outside our current goals; and the fourth is a data-protection goal to which our `protfr` provides an elegant solution.

CAIF inherits many aspects of previous TEEs. The protected code segment is available in SGX, and is featured in Sancus. An intrinsic secret used for key derivation is present in SGX; a shared secret like k_s is also in Sancus. Sancus achieves software-independent assured remote execution of a particular TEE without asymmetric cryptography in the trust mechanism.

However, our `protfr` data escrow is distinctive. The computation of the `protfr` key $\mathsf{kdf}_h($`"pf"`$,\ IS,\ src,\ dst)$, while straightforward, appears not to have been considered in any previous TEE design. The SGX `egetkey` primitive yields nothing similar. The data source obtains a value like $\mathsf{kdf}_h($`"c"`$,\ IS,\ src)$; the data destination, receives $\mathsf{kdf}_h($`"c"`$,\ IS,\ dst)$. These incomparable keys yield no shared secret; so confidential delivery of data between TEEs seems to require asymmetric cryptography.

Allowing confidential delivery between TEEs without any dependence on long-term method asymmetric cryptography is a new contribution of CAIF. We can securely install new digital signature algorithms long after anchoring.

8.2 Ideal functionality methods

Section 4 shows that CAIF, when implemented with strong cryptography, is indistinguishable from the ideal functionality IF. Hence we modeled protocols with a simple, IF-style state-based treatment of the CAIF instructions. Corollary 1 ensures that no tractable observer can tell the difference anyway.

This strategy derives ultimately from Goldreich, Goldwasser, and Micali's work on random functions [18], showing a notion of pseudorandomness to be

indistinguishable from random for any tractable observer. Canetti's classic on Universal Composability [11] confirms its value vividly; it shows how to implement many functionalities securely and justifies their use in all higher level protocols. An enormous literature ensued; our Theorem 1 being a small instance of this trend.

8.3 Protocol analysis

Analyzing security protocols has been a major undertaking since the late 1970s [32]; Dolev and Yao soon suggested regarding cryptographic messages as forming a free algebra and using symbolic techniques [15]. A variety of formal approaches follow them, e.g. [9,29,39]. Computational cryptography also suggests methods for protocol verification [6,10]. This raises the question whether symbolic methods are faithful to the cryptographic specifics, with a number of approaches yielding affirmative results in some significant cases [1,30,4]. Our Thm. 1 also supports the soundness of our symbolic protocol analysis.

Our protocols read and write state. State raises distinct problems from messages. These could be addressed in Tamarin, whose multiset rewriting model is fundamentally state-based [28]. By contrast, CPSA offers state in a primarily message-based formalism [20,38,21]; for its current treatment of state, see its manual [27, Ch. 8]. Squirrel interestingly expresses state computationally soundly [3].

CPSA has, helpfully, two modes. As a *model finder* it computes the set of essentially different, minimal executions [19]. This guides protocol design, showing what is achieved by protocols before they meet their goals. CPSA is also a *theorem prover* for security goals proposed by its user; it produces counterexamples otherwise [40].

9 Conclusion

CAIF offers minimal hardware for services to ensure the *provenance* of data, and to *protect* it for known peer services. Equipped with strong symmetric cryptoprimitives, CAIF provides a secure implementation of an ideal functionality achieving provenance and protection directly.

We designed design a sequence of protocols for CAIF. They start with an initial secure anchoring, a ceremony in a protected space, to establish a secret k_s shared with an authority. By protecting this key and its derivatives by `protfr`, we construct channels to known services on the device. These channels may be used from distant locations, e.g. if the device is on a satellite, to assure remote execution for new programs.

A delegation service using new algorithms can yield trustworthy certificate chains for signing keys usable only by known services on the device. So assured remote execution can outlast the safety of any one asymmetric algorithm.

This is a core need for secure reprogramming: it *authorizes* new programs for remote interactions. A second need for secure reprogramming is a way to

deauthorize programs, blocking rollback attacks, in which the adversary benefits by interacting with an old version. Thus, secure reprogramming also needs irreversible changes, either to prevent the old programs from running, or just to block access to the keys that previously authenticated them. This appears to require little more than monotonic counters and constraints on which services can advance them. But that remains as future work.

References

1. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.
2. Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. Status report on the third round of the NIST post-quantum cryptography standardization process. https://doi.org/10.6028/NIST.IR.8413-upd1, July 2022.
3. David Baelde, Stéphanie Delaune, Adrien Koutsos, and Solène Moreau. Cracking the stateful nut: Computational proofs of stateful security protocols using the squirrel proof assistant. In *IEEE Computer Security Foundations Symposium*, 2022.
4. Gergei Bana and Hubert Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 609–620, 2014.
5. David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Tamarin: verification of large-scale, real-world, cryptographic protocols. *IEEE Security & Privacy*, 20(3):24–32, 2022.
6. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology – Crypto '93 Proceedings*, number 773 in LNCS. Springer-Verlag, 1993.
7. Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. A tutorial-style introduction to DY*. In *Protocols, Strands, and Logic*, volume 13066 of *LNCS*, pages 77–97. Springer, 2021.
8. Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1):1–135, 2016. DOI: 10.1561/3300000004.
9. Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. *ProVerif 2.00: automatic cryptographic protocol verifier, user manual and tutorial.* INRIA, 2018. https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf.
10. R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology—EUROCRYPT 2001*, LNCS, pages 453–474. Springer, 2001.
11. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001. Extended version as of 2020 available at https://eprint.iacr.org/2000/067.pdf.

12. Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. https://arxiv.org/pdf/2303.15540.pdf, March 2023.
13. Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.
14. Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.
15. Daniel Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.
16. Scott L. Dyer, Christian A. Femrite, Joshua D. Guttman, Julian P. Lanson, and Moses D. Liskov. Cryptographically assured information flow: Assured remote execution (report). Arxiv, Feb 2024. https://arxiv.org/abs/2402.02630.
17. William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Issues and requirements. In *19th National Information Systems Security Conference*. NIST, 1996.
18. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, aug 1986.
19. Joshua D. Guttman. Shapes: Surveying crypto protocol runs. In Veronique Cortier and Steve Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, Cryptology and Information Security Series. IOS Press, 2011. https://web.cs.wpi.edu/~guttman/pubs/shapes_surveying.pdf.
20. Joshua D. Guttman. State and progress in strand spaces: Proving fair exchange. *Journal of Automated Reasoning*, 48(2):159–195, 2012.
21. Joshua D Guttman, Moses D Liskov, John D Ramsdell, and Paul D Rowe. Formal support for standardizing protocols with state. In *Security Standardisation Research*, pages 246–265. Springer, 2015.
22. Joshua D. Guttman and John D. Ramsdell. Understanding attestation: Analyzing protocols that use quotes. In *Security and Trust Management*, volume 11738 of *Lecture Notes in Computer Science*, pages 89–106. Springer, 2019.
23. Intel trust domain extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf, Februrary 2022. ID 690419.
24. David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf, October 2021.
25. Ilia Lebedev, Kyle Hogan, and Srinivas Devadas. Invited paper: Secure boot and remote attestation in the sanctum processor. In *IEEE Computer Security Foundations Symposium*, pages 46–60. IEEE Computer Society, 2018.
26. Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20*, pages 38:1–38:16. ACM, 2020.
27. Moses D. Liskov, John D. Ramsdell, Joshua D. Guttman, and Paul D. Rowe. *The Cryptographic Protocol Shapes Analyzer: A Manual for CPSA 4*. The MITRE Corporation, 2023. https://github.com/mitre/cpsa/blob/master/doc/cpsa4manual.pdf.
28. Simon Meier, Cas Cremers, and David Basin. Efficient construction of machine-checked symbolic protocol security proofs. *Journal of Computer Security*, 21(1):41–87, 2013.

29. Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 696–701. Springer, 2013.
30. Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Theory of Cryptography Conference*, pages 133–151. Springer, 2004.
31. Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *ACM Symposium on Theory Of Computing*, pages 427–437, 1990.
32. Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *CACM*, 21(12):993–999, December 1978.
33. Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*, pages 479–498, 2013.
34. Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, July 2017.
35. National Institute of Standards and Technology. FIPS 203 (draft): Module-lattice-based key-encapsulation mechanism standard. https://doi.org/10.6028/NIST.FIPS.203.ipd, August 2023.
36. National Institute of Standards and Technology. FIPS 204 (draft): Module-lattice-based digital signature standard. https://doi.org/10.6028/NIST.FIPS.204.ipd, August 2023.
37. National Institute of Standards and Technology. FIPS 205 (draft): Stateless hash-based digital signature standard. https://doi.org/10.6028/NIST.FIPS.205.ipd, August 2023.
38. John D. Ramsdell, Daniel J. Dougherty, Joshua D. Guttman, and Paul D. Rowe. A hybrid analysis for security protocols with state. In *Integrated Formal Methods*, pages 272–287, 2014.
39. John D. Ramsdell and Joshua D. Guttman. CPSA4: A cryptographic protocol shapes analyzer. The MITRE Corporation, 2023. https://github.com/mitre/cpsa.
40. Paul D. Rowe, Joshua D. Guttman, and Moses D. Liskov. Measuring protocol strength with security goals. *International Journal of Information Security*, February 2016. DOI 10.1007/s10207-016-0319-z, http://web.cs.wpi.edu/~guttman/pubs/ijis_measuring-security.pdf.
41. Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. SoK: Hardware-supported Trusted Execution Environments. https://arxiv.org/pdf/2205.12742, May 2022.
42. Victor Shoup. A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, 2001. https://eprint.iacr.org/2001/112.pdf.