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 to known processes (i.e. processes executing known code) running on that device. Assured remote execution requires a hardware basis including cryptographic primitives.**

We give a simple hardware-level mechanism called *Cryptographically Assured Information Flow* (CAIF) to enable Assured Remote Execution. CAIF is akin to existing Trusted Execution Environments, but securely implements an ideal functionality for *logging* and *confidential escrow*.

CAIF achieves assured remote execution, and symbolic 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 a core part of secure remote reprogramming. [1]

I. 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 grid. Long-term, can you deliver messages exclusively to specific, known processes executing on d? Can you, when receiving a message, be sure it was prepared by a specific, known process on d? Can the processes run code written and delivered long after d was initialized?

This is the *Assured Remote Execution* challenge.

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. Thus, solving the assured remote execution challenge requires both device-local mechanisms on d and distributed mechanisms to coordinate d with its owner or peers. This paper offers a device-local mechanism in §§ III–VI, and shows that it suffices for protocols to coordinate with d starting in § VII.

Good solutions should:

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, where we identify processes by the hash of their executable code and constants. This allows a process on d receiving certain data values to identify the processes that generated them, and it allows a process on d wanting to pass a data value confidentially to a particular recipient process to do so without any other process observing the value. We call it CAIF, for *Cryptographically Assured Information Flow*.

Our hardware basis uses only hashing, 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 or installs their own. Hence, higher level software can use CAIF to help prevent malicious execution without circular dependencies. So our adversary model meets criterion 2.

An ideal functionality [20], [11] characterizes CAIF, and our cryptographic mechanism *simulates* it to within negligible probability (Cor. 1). Lemmas saying the ideal functionality enforces our intended information flow constraints are thus also near approximations to the cryptographic mechanism.

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 [42]. 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.

Corresponding author: Joshua Guttman, `guttman@mitre.org`.

One enables a *service* (certain processes) to *log* itself as the source or authorizer of a data item. Other parties can subsequently *check* whether an expected service has logged a data item. The other function enables a service to *escrow* a piece of data for a service as recipient. Only that recipient can then *retrieve* it. In a *check* or *retrieve* operation, the recipient of a logged or escrowed value names the value's expected source. The operation fails if that source did not *log* or *escrow* it, so success guarantees the value's provenance.

The cryptography-free ideal functionality of § III implements these functions via an *unbounded secure memory*, proving desirable behavioral properties (Lemmas 1–3). This secure memory holds the logged associations of service and data, and the escrowed associations of data, source, and recipient.

CAIF devices (§ V) use MACs for logging and authenticated encryption for data escrow. They require only one unshared "intrinsic secret" IS, used as an input to key derivation for those cryptoprimitives. CAIF devices name services by the hash of their executable code, ensuring that two services with the same name will have the same computational behavior.

Assured remote execution requires evidence a service svc on d sent messages m. Our scheme uses a signing keypair (sk, vk) with a certificate chain for the verification key vk. The certificate chain provides public evidence that sk is escrowed for svc, and that the provenance of sk leads back to a previous service svc_0 that escrows it *only* for svc. At the root of the chain, a certifying authority authenticated d using a shared key that was established early in d's life.

If m is signed with sk, then svc bears responsibility (see IX-A). For confidential channels to svc, the signed messages m can be used for key encapsulation [46].

Contributions. We make three main contributions.

1. We define CAIF and its ideal functionality for data logging and escrow (§§ III, V).
2. We prove that CAIF, if using strong cryptography, is computationally indistinguishable from an instance of the ideal functionality (§ VI).
3. We develop a sequence of protocols on CAIF to achieve assured remote execution. Symbolic protocol analysis shows they achieve this goal despite a strong adversary that can execute code of its own choice on our devices. § VII gives our strategy, and §§ VIII–IX provide details.

CAIF's guarantees are independent of delicate systems-level considerations, such as how software obtains control at boot. § II identifies key challenges and a use case, which § IV shows how to meet. §§ X–XI discuss related work and conclude.

II. Current challenges

CAIF is motivated by several ingredients in the current situation for cryptographic devices and secure systems design.

A. Background challenges

The quantum-resistant transition. Motivated by the quantum cryptanalytic threat, new quantum-resistant primitives are now in draft standard [2], [38], [39], [40]. However, CAIF's guarantees are independent of asymmetric cryptography such as digital signatures. Long-lived CAIF devices meet their guarantees even if these primitives are broken and revised, or if their key sizes must be adjusted. New asymmetric algorithms and root-of-trust keys—typically, signature verification keys—can be installed securely on geographically dispersed CAIF devices, yielding a long-term security architecture depending only on stable, efficient symmetric cryptographic primitives.

Trusted Execution Environments. If asymmetric algorithms may need to evolve, existing Trusted Execution Environments (TEEs) [14], [25], [26] are not the right tool. Although they use only symmetric cryptography at the hardware level, they rely on public key encryption to protect data passing from one enclave to another. This was acceptable when quantum cryptanalysis 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, and does not allow us to construct it: Some enclaves may choose to disclose their key, producing counterexamples to Lemma 1 (cf. § X-A). Second, CAIF's escrow also constructs keys for ordered *pairs* of services, and uses those for its confidential escrows. This provides a symmetric method for service-to-service information flow, which existing TEEs entirely lack.

B. A CAIF application: Satellite reprogramming

CAIF pays off for widely dispersed, long-lived devices with clear security goals and programs that may need to evolve.

For instance, the owner of a network of communications satellites needs to manage them securely from the ground through decades of use. The satellites must also create secure connections among themselves. Customers and other network providers also need secure connections with them.

These secure connections may need updated asymmetric cryptography within the lifetime of the satellites as quantum resistant algorithms or key sizes may evolve. New crypto code with new root-of-trust keys must then be installed while the satellites and their crypto hardware are aloft. Data integrity is critical: An adversary whose bogus root key is accepted can command the satellite. Thus, services can use a root key only if it must have been installed by the authorized local service, which in turn authenticated its origin from the satellite's management on the ground. Local provenance leads back to a root key installer, from which an authenticated protocol must lead back to the management.

Updating signature algorithm and root-of-trust key. Suppose a CAIF-equipped device d, after some initial terrestrial preparation, called "anchoring," is loaded into a satellite to provide cryptographic functionality and lofted into orbit; the operator of the satellite wants to exert control over the satellite via d. This operator is called the *device authority DA*. We assume for now that *DA* shares a long-term secret k_{ar} which can be used only by a particular service with code hash *arh*; § VIII-C sets up k_{ar} via the terrestrial preparation.

Years later, the signature algorithms d uses may need updating; maybe the old ones are already compromised. *DA* can send new code $svAlg$ for signature verification to d via ordinary communications, but must also send evidence the new code is trustworthy and received unaltered, as well as a new root-of-trust signature verification key vk for checking signatures with $svAlg$. In § IV we show how to use CAIF to do so securely.

III. An Ideal Functionality for CAIF

We characterize CAIF 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 V we will introduce CAIF devices using cryptography, and in Section VI we prove that these CAIF devices offer a near approximation to the IF's behavioral properties.

A. 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 hash serves 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. Logging 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 previously logged v into atlog via iattest, with tag τ. Otherwise, it returns *false*.

Any active process can use icheck to see if 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. Some function F_{log} of P_s and v determines τ. To implement iattest, one would use a MAC as F_{log}, so τ is the MAC tag. The tag τ and v may be passed from source to recipient through shared resources such as a file-system.

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 just 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 randomized *handle* η. It 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 looks up the index (η, P_s, P_r) in the table protstore. If any entry $(\eta, P_s, P_r) \mapsto v$ is present in protstore, that v is returned to P_r. Otherwise, it fails.

Lemmas 2–3 below depend on iprotect's choice of handles η. We assume it samples η from a distribution D_{j,P_s,P_r} for data of length j escrowed by principal P_s for recipient P_r where:

(i) the randomized choice of η is independent of which v is presented, for all v of a given length j;

(ii) For different lengths $j \neq j'$, the supports $\mathsf{supp}(D_{j,P_s,P_r})$ and $\mathsf{supp}(D_{j',P'_s,P'_r})$ are disjoint; and

(iii) The IF does not re-use any handle η; it checks the table entries and re-samples in case of collision.

B. 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: 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. Lemma 1 makes no claim about sequential order; *check* confirms only presence not sequence relative to other events. Values v containing hash chains can add sequential information as usual.

Properties of IF: Protection. The analog of Lemma 1 holds for $\mathsf{iprotect}$, using the re-sampling assumption (iii):

Lemma 2: 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. IF leaks *no information* about the values associated with handles that are never retrieved.

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 events in α.

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

Lemma 3: 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 (ii), 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. Lemma 3 is strong; since CAIF must approximate it using concrete cryptography, it achieves only a computational approximation to it. 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 $\text{IF}[F_{log}, \{D_{\ell,P_s,P_r}\}]$. The lemmas hold for all values of these parameters satisfying (i)–(iii).

IV. Using the CAIF functionality

A. Satellite reprogramming via CAIF

We turn back to our satellite reprogramming challenge for CAIF from § II-B. We use the assumed shared secret k_{ar} as a MAC key to authenticate messages from *DA*. MAC suffices because we need no confidentiality here; integrity, authentication, and authorization are the goals.

On d, an *authorized recipient* has service identity or code hash *arh*, and a *signature verifier* with code $svAlg$ has code hash $svh = hash(svAlg)$.

Authorized recipient with hash *arh*: it receives an incoming containing a code hash svh and a signature verification key vk and MAC-checks it using k_{ar}. On success, it $\mathsf{iattests}$ a *verifier record* containing vk and svh and also $\mathsf{iattests}$ a *client record* containing svh. The verifier record authorizes svh to use key vk. The client record authorizes clients to use svh to verify signatures.
The service *arh* is trusted to make these authorization claims. As *arh* is independent of the signing algorithm, it may be installed at device initialization and left permanently unchanged.

Signature verifier with hash $svh = hash(svAlg)$: it obtains a verifier record containing a verification key vk and its own hash svh. If the record $\mathsf{ichecks}$ as logged by *arh*, then svh awaits requests (m, σ) from clients. If the signature verification succeeds for (m, σ) with key vk, then the service $\mathsf{iattests}$ a confirmation record containing yes, m, and σ. On failure, it $\mathsf{iattests}$ an error record.
The authority *DA* compiles the constant *arh* into the code $svAlg$. Altering the constant alters the hash svh, so a log record with provenance from svh ensures the right origin *arh* was checked.

Clients, when receiving a purported signed message (m, σ), obtain a client record containing a hash value svh, using icheck to ensure it was logged by *arh*. They then request svh to verify the signature. When a confirmation record containing yes, m, and σ is received, and it $\mathsf{ichecks}$ successfully for svh, the signature is valid. The clients' code also embeds *arh* as a constant.

If *DA* authorizes *only* verifier code acting as described, then messages will be accepted only if validly signed. Symbolic protocol analysis confirms this, assuming k_{ar} is protected from compromise at *DA*.

The symbolic protocol analysis makes some fine points explicit. For instance, *DA* authorizes more than one root verification key vk for the code svh, the client does not learn which key verified a particular (m, σ). Adding vk to the confirmation record achieves this, if desired.

A remote authority can use this method to enable a CAIF device to use new signature verification algorithms coded, delivered, and authorized long after CAIF device became physically inaccessible. An initial anchoring event made the shared secret k_{ar} available only to *arh* (see § VIII-C).

Deauthorizing an old, no longer secure signature algorithm, by contrast, requires an irreversible change, lest an adversary roll d back to it. A monotonic counter, judiciously used, would likely suffice, but that remains as future work.

Assured remote execution? This protocol does not offer assured remote execution: It does not deliver evidence to a remote observer that any svc is active on d. It offers local client services information provenance for vk and svh, namely evidence they were obtained by *arh*, and are thus authorized if *DA* safeguarded k_{ar}. the scheme authenticates messages from *DA* to d assuming *DA* protects its signing key.

Assured remote execution offers a converse, i.e. authenticating messages from a service svc on d to external peers. *DA* can use shared keys like k_{ar} for this authentication. *Signature delegation* (§ IX) subsequently extends this svc-to-*DA* assured remote execution to allow any peer to authenticate any service on d using (possibly new) digital signature algorithms. The underlying message authentication allows d to get signing keys for these services certified.

Access to long-term secrets such as k_{ar} at the *DA* should be rare and stringently restricted. We use them only to authorize new asymmetric algorithms on the device or to certify keys.

CAIF for assured confidentiality. Signature verification keys require integrity; by duality signing keys require confidentiality. A signing key that will authenticate a service on d must rely on the signing key remaining confidential; adversary-installed code must not hijack the key. Thus, local flow should carry a *DA*-certified private signing key only to the code authorized to use it. *DA* must know that the private signing key cannot reach any further code as recipient. We use the CAIF escrow operation iprotect to achieve this; see § IX.

Another application: critical infrastructure. Devices controlling electric grids, water systems, etc. have succumbed to widespread infiltration, unsurprisingly as their software is very hardware-specific. As they are long-lived and geographically dispersed, hands-on reprogramming is impractical. Remote methods are needed to guarantee new code controls them. Preliminary work with Field Programmable Gate Arrays suggests CAIF's hardware burden is modest, making it feasible for these small, cost-constrained devices.

B. Techniques for effective CAIF use

A few core ideas underlie effective CAIF use.

Local chains of provenance. Multistep chains of provenance can use CAIF logging or escrow. Suppose svc_0 logs m_0 that says, "I computed v_0 from v_1, which was logged by svc_1." Local services can icheck the m_0 log entry, ensuring svc_0 logged it. If the svc_0 code does not log m_0 unless it ichecks v_1 and computes v_0, then svc_1 logged v_1. If v_1 is a message m_1 of similar form, we can follow the chain backward.

The inference requires knowing what svc_0's code may do. CAIF does not help proving this, but given evidence of it, CAIF yields runtime conclusions about v_0 and v_1's provenance.

Confidential intermediate values delivered exclusively to svc_1 via escrow may also be chained together (§ VIII-B). The provenance information assures local services that the flow occurred, with *some* confidential value v_1 escrowed with a handle η_1.

How to use CAIF's local guarantees remotely. Anchoring (§ VIII) and signature delegation (§ IX) yield authenticated messages from known services svc on d. If svc reports locally validated provenance chains on d, the peer learns the source of svc's data, such as the pedigrees of secret keys.

But: Hashes are unpredictable. When one service escrows confidential data for a peer service, how does the source choose the hash of its target service? And how does a service decide which hash stands for an authorized source service from which it should accept data? Hashes are fragile and do not reflect semantics. There are a number of approaches:

1. The hash h_p of the intended peer service is embedded in the active service's executable code. Alterations to h_p in the code change its code hash h_a to some other h'_a, rendering data such as keys escrowed for h_a unavailable. If the peer is compiled first, the constant h_p can appear in h_a's executable. Alternatively, several services may be constructed jointly with each others' hashes [12].
2. The hash h_p may be found within a logged or escrowed record r the service reads, where CAIF logged h_0 as the origin of r. If h_0 is known to be trustworthy for a given purpose, e.g. by method (1), then trust for related purposes may be delegated to h_p by r.
3. The hash h_p may be received in a message authenticated as coming from an authority such as the device authority *DA*, who may delegate trust for a particular purpose to the service with hash h_p.
   The incoming message may be authenticated using a key k found in an escrowed record with source hash h_0; k's trustworthiness then comes via method (2). Alternatively, an exchange occurring only in a protected environment can also authenticate the incoming message.
4. The hash may be supplied as a parameter in an argument vector or in an unauthenticated incoming message. A hash h_p of bad code may be supplied. This does no harm if h_p is included in messages or attested records generated by the service, and the subsequent recipients can decide whether to trust h_p.

In methods (1) and (3), the authority has had an opportunity to appraise whether the code with hash h_p satisfies some behavioral property. With method (4), the recipient adapts its responses to what it knows about that code.

We used method (1) in § IV-A. Method (2) appears in the client code of § IV-A and makes frequent appearances subsequently. We use method (3) repeatedly in §§ VIII–IX, receiving hashes h in authenticated messages where the authentication depends on escrowed keys. Our anchoring protocol (§ VIII) uses method (3) where the authentication depends on a protected environment, which provides the fundamental trust basis for our assured remote execution protocol.

Many combinations of the four methods are useful.

In our analyses below, we express behavioral properties as protocol roles sending and receiving messages, and loading and storing records into device state records. These roles define what we are trusting the programs to do, namely to act on data values of the relevant forms only in the patterns codified in the roles. Their behavior may vary in other respects without undermining our conclusions. Creating trustworthy software is hard, but a precise description of what the software will be trusted to do (and avoid doing) helps.

Some limits. The benefits of CAIF apply only to devices satisfying the specification in § V. Hence, manufacturers must prevent back doors; customers need reliable supply chains from the manufacturer. The customer must also anchor it properly (as in § VIII), and protect the anchoring shared secret.

For different organizations' CAIF devices to cooperate, each needs to know the peer organization's devices are CAIF-compliant and were properly anchored. Different network providers with CAIF-equipped satellites can interoperate securely based on business agreements; interoperation among allies is also reasonable. In a permissioned blockchain, the controlling authority can make contractual agreements with the parties. Open access blockchains, by contrast, have no way to check which devices are in fact CAIF devices.

Information flow. The CAIF mechanism allows services to determine *from which* service some data has come, and *to which* service it may be delivered. In this sense, it controls information flow. However, it has different goals from information flow in the sense of the large literature descending from Goguen and Meseguer [19]. In particular, rather than enforcing a system policy, our mechanism allows services to enforce their own application-specific policies; it is not a mandatory mechanism but a discretionary mechanism. Moreover, unlike classical noninterference, these policies are not "transitive;" service s_2 may accept data from s_1 and deliver it to s_3, whereas s_1 and s_3 may refuse direct flow [44]. Thus, CAIF enables discretionary, non-transitive information flow control.

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 $\mathit{index} \mapsto \mathit{result}$. 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. $\mathsf{supp}(D) = \{x \colon 0 < Pr[y \leftarrow D; y = x]\}$.

V. 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 (§ V-A); and, second, two pairs of *instructions* or basic operations to ensure provenance and control access of data passed among services (§ V-B). Auxiliary operations are also needed to manage services (§ V-C). § V-D summarizes what being a CAIF device requires.

A. CAIF control over 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*.

B. CAIF instructions

CAIF offers two pairs of instructions for 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. The primitive operations use symmetric cryptography. Their keys are derived from one or more service hashes and a device-local unshared secret called the Intrinsic Secret.

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. Hardware design can help prevent *IS* being accessible except for key derivation. It may be implemented as a set of fused wires as in SGX or as a Physically Unclonable Function as in Sanctum [27].

The hardware furnishes four cryptographic primitives, namely a key derivation function kdf_h, a Message Authentication Code mac_h, and an authenticated symmetric encryption enc_h with the decryption dec_h. Thus, when c was not generated by an invocation of $\mathsf{enc}_h(p, k)$, then $\mathsf{dec}_h(c, k)$ is negligibly likely to return a non-failure result.

A conceptual view of the cryptographic hardware components is in Fig. 1. Triple arrows are buses; 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 for *attesting locally* and *checking an attestation*; they identify a service that has generated or endorsed a data

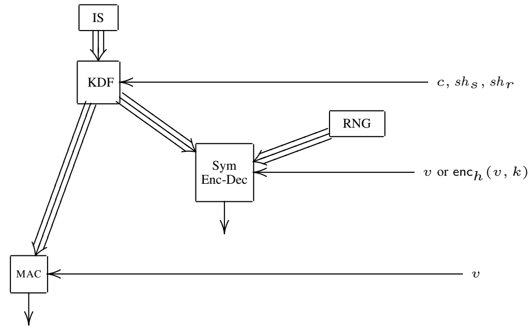

Fig. 1. CAIF Hardware Cryptography Components

value v, typically the data in a region of unshared heap defined by a pointer and a length.

`attestloc`(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 makes m available. The MAC tag m and v may subsequently be copied anywhere.

`ckattest`(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 $m = \mathsf{mac}_h(k, v)$.

A service may thus log itself as the origin or approver of v; any recipient of v and m, if executing on the same device with the same intrinsic secret IS, can subsequently ascertain its provenance. More specific "intent" may be encoded into the content of v, which may then be copied to a shared resource, e.g. a filesystem. Thus, `attestloc` and `ckattest` provide a device-local mechanism for asserting and confirming provenance.

Protection and provenance via encryption. The remaining primitive operations *protect* a value *for* a named recipient by authenticated encryption, and *retrieve* a value *from* a named source by decryption. The plaintext should always be located within unshared heap. The ciphertext can pass freely through shared resources, but only the stipulated recipient recovers its content, and only if it was from the expected source.

`protfor`(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 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.

`retrvfm`(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 hash sh_s, and the code hash sh of the current service. Fails if the (authenticated) decryption fails, i.e. if the associated tag is wrong, 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 $v = \mathsf{dec}_h(e, k)$, the plaintext v is stored back into unshared heap.

Local provenance with access control results from `protfor` and `retrvfm`. Only the service specified in the `protfor` learns anything from the result, and only on the same device. If sh or IS differs, the k in `retrvfm` will differ, causing the authenticated decryption to fail. If sh specifies the wrong source sh_s, k will again differ and failure ensue.

C. 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-services`.

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

Any binary executable may become a service, with or without CAIF instructions; that binary executable is then indelibly associated with it through the service hash. Thus, its identity is correctly reflected however it uses the four core CAIF instructions. Varying implementations can use different versions of the auxiliary instructions.

D. CAIF devices

By a *CAIF device* we mean a hardware device that enables the CAIF instructions in § V-B to be performed by services of the kind given in § V-A; the auxiliary operations in § V-C 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 R_k using up to q queries to kdf_h and a polynomial number of operations relative to the security parameter k. See [17].

MAC: The `attestloc` and `ckattest` 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 `protfor` and `retrvfm` primitives use an enc_h secure against chosen ciphertext attacks [34]. 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.

Code hashing for services should be second preimage resistant.

CAIF devices thus form families, parameterized by a security parameter k. § VI argues that as k increases and the advantages just described decrease, CAIF devices become indistinguishable from instances of IF.

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 as 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 `protfor` 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 [28] suggests a provisional way to enforce the memory protection in § V-A, 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.

VI. 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, as defined in § V-D.

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 the successive (σ_{j+1}, r_{j+1}) in the distributions $\delta(\sigma_j, q_j)$.

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 i^{th} position of $\mathcal{S}$.

A. Defining advantages

A CAIF device is an oracle with unchanging state, the intrinsic secret IS, with constant length $|IS|$. Its commands are the four instructions of § V-B, 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 parameterized by the security parameter k, with cryptographic primitives yielding advantages defined in § V-D.

An instance of the IF is also an oracle. Its state consists of atlog and $\mathsf{protstore}$. The transition relation δ is defined in § III-B using F_{log} and the family D_{ℓ, P_s, P_r}. We identify the instruction `attestloc` 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 § III, 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:

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

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

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

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 $\mathsf{Adv}_{\textit{p-c}}(A, C, k, q, t) = |p_0 - p_1|$ where $p_b =$

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

B. Two lemmas

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, $\mathsf{Adv}_{\textit{a-u}}(A, \text{CAIF}, k, q, t) \leq$

$$q\mathsf{Adv}_{\textit{eu-mac}}(\Gamma_{1,q}(A), k, q) + \mathsf{Adv}_{\textit{prf}}(\Gamma_2(A), k, q+1) \qquad ///$$

Proof: See [17] for 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, $\mathsf{Adv}_{\textit{p-u}}(A, \text{CAIF}, k, q, t) \leq$

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

C. Proving secure implementation

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

$F_{log}(v, P) = \mathsf{mac}_h(sk, v)$ where $sk = \mathsf{kdf}_h(\texttt{"at"}, \mathit{IS}, P)$;

D_{ℓ,P_s,P_r} is the distribution generated by $\mathsf{enc}_h(0^\ell, sk)$ where $sk = \mathsf{kdf}_h(\texttt{"pf"}, \mathit{IS}, P_s, P_r)$.

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

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

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 the random function 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, existentially unforgeable mac_h, collision-resistant code hashing, and IND-CCA2 enc_h, then that CAIF device is computationally indistinguishable from an instance of IF. ///

VII. Assured Remote Execution Strategy

In §§ VII–IX we verify protocols for assured remote execution on CAIF against a code-selecting adversary. Each device has a uniquely identifying name, its *immutable identifier* $imid$; $imid$s, like other names, may be publicly known.

When we speak of *protecting* a value *for* a service or *retrieving* a value *from* a service, we mean that the value is the argument to `protfor` or the result of `retrvfm`, resp.

A. Achieving Assured Remote Execution

Our strategy uses a succession of steps, and does not depend on any digital signature algorithm to remain secure throughout a device's lifetime. We start from an assumption. The device must be started running a known program once, at the beginning of its lifetime. The manufacturer—already trusted

$$\mathsf{Adv}_{ind}(A, \textsc{caif}, k, q)$$

$$\Pr[A^{\mathcal{O}_0(\cdot)} = 1] \qquad \Pr[A^{\mathcal{O}_1(\cdot)} = 1] \qquad \Pr[A^{\mathcal{O}_2(\cdot)} = 1] \qquad \Pr[A^{\mathcal{O}_3(\cdot)} = 1]$$

$$q^2 \mathsf{Adv}_{indCCA2}(\Gamma_{7,q}(A)) \qquad \mathsf{Adv}_{prf}(\Gamma_8(A)) \qquad q[\mathsf{Adv}_{a\text{-}u}(\Gamma_{9,q}(A)) + \mathsf{Adv}_{p\text{-}u}(\Gamma_{10,q}(A))]$$

Fig. 2. Implementation theorem proof strategy: Lemmas 4 and 5 bound the last term

to produce correct hardware—initializes it with a compliant service to run first. This *anchor* service must run *only* in a secure environment. Devices undergo a state change—such as fusing a wire, flipping a switch, or advancing a monotonic counter—to prevent re-execution after successful anchoring.

The owner or authority controlling this device, its *device authority DA*, anchors the device, sharing the secret k_s with it in step 1. The *DA* must store k_s securely for later use.

Starting with our assumption:

0. We assume *local* execution of a single service, the *anchor service* 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 recipient service svc_1 via `protfor` in shared storage. It then zeroes its unshared memory and exits (§ VIII).
   A state change prevents rerunning the anchor service.
2. svc_1 is a *symmetric key distributor service*. It will receive authenticated requests from *DA*; each request specifies a service hash sh, and svc_1 derives a secret $\mathsf{kdf}(k_s, sh)$ that it protects for the exclusive use of the service with hash sh in shared storage. It then zeroes its unshared memory and exits (§ VIII-C).

For instance, the key k_{ar} of § IV-B is derived by the symmetric key distributor service as $k_{ar} = \mathsf{kdf}(k_s, arh)$.

To escape from using shared secrets to infer assured remote execution, we establish a *signing key delegation service*. Possibly long after the time of initialization, possibly repeatedly, § IX installs new programs and authorizes them using a key from the symmetric key distributor:

3. A *set-up service* with hash suh generates a device signature key pair $(\mathsf{dk}, \mathsf{dvk})$. Using the shared secret $\mathsf{kdf}(k_s, suh)$ to authenticate, it proves to a certifying authority CA operated by *DA* that it holds the signing key dk. A certificate 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 service dsh exclusively. It then zeros its unshared memory, and exits.
4. The *delegation service* has hash dsh. When invoked with a target service hash sh, it 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 service sh exclusively, in shared memory. It then zeros its unshared memory, and exits.

If sh does not expose dk, then messages signed with sk must come from sh, as required for assured remote execution.

Assured remote execution. Each of these steps builds additional assured remote execution power. Step 0 simply assumes local execution of the anchor service in secure initial environment. Step 1 assures remote execution only for *one* service svc_1, with evidence useful only to *DA*, which holds k_s. Step 2 gives *any* service sh a secret, but evidence of sh's remote execution is useful only those sharing the secret.

Step 4 completes the progression. *Any* sh can receive a signing key documenting its execution on d, and any principal willing to trust the CA's certificate can use the evidence. No shared secrets are needed as sh executes.

Alternative for short-lived devices. If d will never outlive a signing algorithm and keysize, a simpler protocol is enough. The set-up service of step 3 becomes the anchor service. It obtains a certificate on dvk. The signing key delegation service of step 4 provides signing keys to services sh. No shared secrets are needed; however, if algorithms or root keys are superseded, assured remote execution cannot be reestablished. Other benefits of local flow control remain.

B. Compliant roles and adversary roles

Over time, many active processes will execute and use the CAIF instructions on a device. Many of the processes may behave unpredictably, 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. We call them *adversarial services* or just *wildcats*.

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 §§ VIII–IX. 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 or *wildcats* use the CAIF instructions freely, but use service hashes not assumed `compliant`.

Wildcat roles. We specify the adversary's local powers—beyond the usual network 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 `ckattest` 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 `protfor` etc. as desired. However, if a program has a 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 e.g. repeated anchoring (step 1).

VIII. Anchoring a CAIF Device

When a new CAIF device reaches the buyer's warehouse—or alternatively, just before shipping—it is *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, thereby delivering a shared secret k_s. Subsequently the device and *DA* use k_s and derived keys 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
$$imid, anch, dh, n, r,$$
dh being the hash of the destination service meant 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 `protfor` to protect k_s together with the identities $\langle dh, anch\rangle$ for service *dh* exclusively into shared storage; it zeroes its unshared memory and exits.

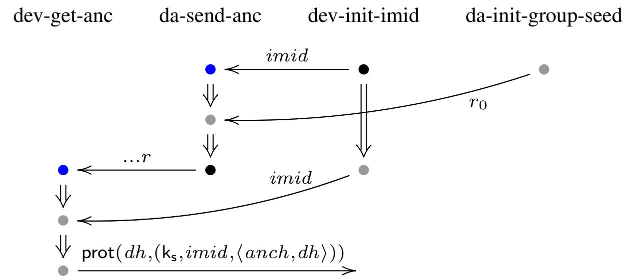

Fig. 3. Metal room activity, symmetric anchoring

Hence, k_s is available on *imid* only to the service with hash *dh*. To ease managing long-term secrets, r may itself be derived by differentiating a group seed r_0 for *imid*, setting $r = \mathsf{kdf}_h(r_0, imid)$, so $k_s = \mathsf{kdf}_h(\mathsf{kdf}_h(r_0, imid), imid)$.

The anchor service must run *only* with a secure metal room channel, requiring an irreversible state change after which *anch* will no longer run, e.g. a switch being flipped.

After the ceremony in the metal room, k_s remains permanently available to *dh* to secure later remote interactions.

A. 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 roles represent the *DA*'s interaction with the device. One does set-up to manage the shared secrets, creating a *group seed* r_0, from which secrets r are derived. A second *sends the anchor* secret r 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 its derivatives remain unavailable to the adversary.

What happens at the metal room layer 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 r_0 (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 (`protfor` 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 § IX-D.

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

B. Trust chains

We keep track of "chains of provenance" 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 `protfor` records. We also deliver trust chains in messages between parties, who can check them to trace provenance back to the anchoring program.

Trust chains are lists of service hashes. If *trch* is a trust chain, we write the effect of pushing a new hash h to it as $h :: \mathit{trch}$, using a *cons* "::" operation.

Each of the services we specify, when retrieving a trust chain, tests that its code hash is the first and the source service's hash is second. Thus, it has confirmed by the successful `retrvfm` that the prior entry also represented its identity correctly. When extending a trust chain, it pushes its intended recipient's hash to the front of the list.

If an observer, seeing the trust chain, knows that the front n entries are *compliant* roles we have specified, it follows that the $n+1$st entry is the actual source of the record retrieved by the nth entry. If that hash is also known to be compliant, this process can repeat.

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

C. The symmetric key distributor service

The anchor service may be used with any dh. A useful dh is a *symmetric key distributor service*, whose hash we will write skdh. The symmetric key distributor program retrieves k_s packaged with the trust chain $\mathit{trch} = \mathit{skdh} :: \mathit{anch} :: \langle\rangle$, checking its hash skdh and its source anch. It then uses k_s to decrypt a message from the *DA*. This message should contain:

- a target service hash tgth for which a new symmetric key k should be derived;
- a trust chain trch';
- a payload payld to pass to the service tgth when it runs.

The distributor service checks that $\mathit{trch}' = \mathit{trch}$. If so, skdh derives the key $k = \mathsf{kdf}_h(k_s, \mathit{tgth})$. It protects an extended trust chain, the payload, and this key k for tgth:

$$\texttt{protfor}\ \mathit{tgth}\ (\mathit{imid},\ (\mathit{tgth} :: \mathit{trch}),\ \mathit{payld},\ k)$$

into shared storage, after which it zeroes its unshared memory and exits. On device imid, k can only by obtained by the service tgth. Since the *DA* can compute $k = \mathsf{kdf}_h(k_s, \mathit{tgth})$, k enables the *DA* to create an authenticated, confidential channel to imid where only service tgth can be the peer.

D. 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 tgth.

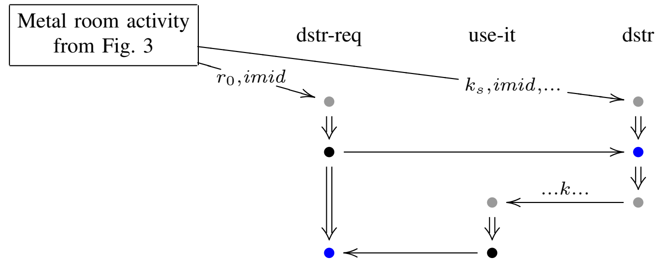

Fig. 4. Distributor request

The analysis for `compliant` code hashes for anch, skdh, and the target service tgth appears in Fig. 4. The distribution service uses k_s generated from r_0 given in Fig. 3; it handles the request and derives k, protecting it for tgth. 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.

If tgth 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 must come after the anchoring ceremony. But because no arrows point *into* the box at the upper right, no part of the remainder has to come before the anchoring ceremony. Assuming that the values stored in state records persist, 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.

IX. 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 $(\mathsf{sk}, \mathsf{vk})$;
2. It emits a certificate-like message signed with dk that binds the new verification key vk to sh and the device imid and additional information. We call this a *delegation certificate*.
3. It protects the signing key sk and additional information for the exclusive use of service sh, placing it in shared storage. It then zeroes its unshared memory and exits.

If a recipient knows the delegation service generated the delegation certificate and observes a message signed with sk, it can infer sh has been active on imid. To ensure the delegation service generated the certificate, we (in turn) use a *delegation set-up service* to get a *CA* certificate for dk. Call the delegation service's hash dsh and the set-up service's hash 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 $(\mathsf{dk}, \mathsf{dvk})$ 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* into shared storage; it zeroes its unshared memory and exits.

The additional information mentioned includes trust chains asserting full data provenance (§ VIII-B).

A. Assumptions and security goal for delegation

Our delegation scheme relies on the assumptions:

(i) the CA is uncompromised;

(ii) the CA receives the proof-of-possession from *suh* on *imid* on an authenticated channel when certifying dvk.

The anchoring in § VIII will provide the authenticated channel 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 one observes: A digital certificate m_1 from CA binding dvk to *dsh* on *imid*; a delegation certificate m_2 binding vk with *sh* on *imid*; and a message $m_3 = [\![m_0]\!]_{\mathsf{sk}}$ verifying under vk.

Then 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 m_1 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 m_2 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_3.

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.

B. 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 $(\mathsf{dk}, \mathsf{dvk})$ plus an expanded trust chain $trch' = dsh :: suh :: trch$ via `protfor`. The delegation service then uses `retrvfm` to recover the key pair $(\mathsf{dk}, \mathsf{dvk})$ and $trch'$. When generating a key pair $(\mathsf{sk}, \mathsf{vk})$ for a target service *sh*, it protects $sh :: trch'$ and $(\mathsf{sk}, \mathsf{vk})$ for *sh* with `protfor`, and emits a delegation certificate

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

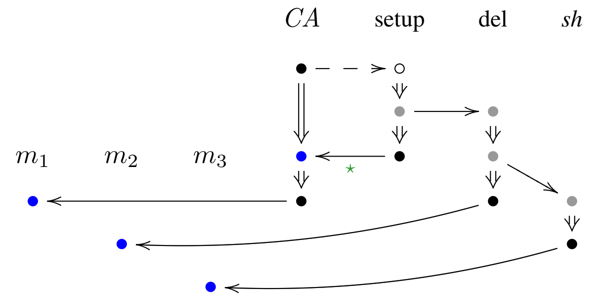

Fig. 5. Message m_3 signed with delegated key; $\star$ authenticated channel

C. Delegation analysis

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 § IX-A. 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 [46].

D. Adapting delegation to the anchor

Anchoring provides a concrete realization of the authenticated channel for the proof-of-possession. The symmetric key distributor (§ VIII-C) generates $k_{sud} = \mathsf{kdf}_h(k_s, suh)$; *suh* uses it to encrypt the proof-of-possession to the CA.

The symmetric key distributor receives a payload *payld* from the *DA* when deriving a key, which it protects for

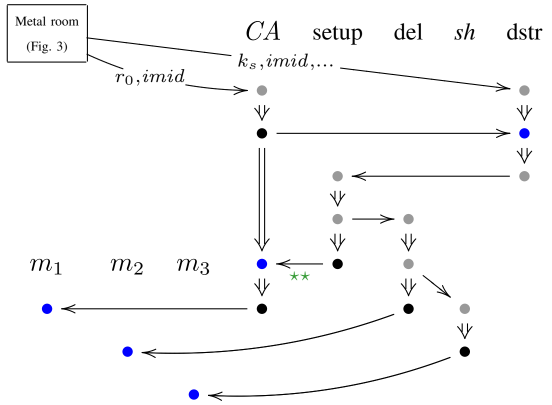

Fig. 6. Message m_3 signed, with anchoring and ⋆⋆ encrypted

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 `retrvfm`, 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*, *skdh*, *suh*, and *dsh* are all in *trch*, the *CA* emits a certificate only when the trust chain is acceptable. We consider the case that they are all compliant.

If the target service hash *sh* is compliant, analysis yields a single possibility shown in Fig. 6 that enriches the Fig. 5. The proof-of-possession *pop* is encrypted under k_{sud} on the ⋆⋆ arrow from *suh* to *CA*. If *sh* may not be compliant, a wildcat retrieve role may alternatively extract and disclose 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 here only in our accompanying text.

X. Related Work

A. Trusted Execution Environments

CAIF is a kind of TEE, like Intel's SGX and TDX [14], [25], AMD's SEV [26], and research such as Sancus [36], [37] and Sanctum [15] and Keystone [28] for RISC-V. Schneider et al.'s recent survey on TEEs and their implementation choices [45] identifies four main security properties for TEEs (p. 1). One is a weak, launch-time version of assured remote execution; the second covers our address space requirements (§ V-A, 1–2); the third, concerning trusted IO, lies outside our current goals; and the fourth is a data-protection goal to which our `protfor` 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 `protfor` data escrow is distinctive. The computation of the `protfor` key 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 kdf_h(`"c"`, *IS*, *src*); the data destination, receives 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 asymmetric cryptography is a new contribution of CAIF, enabling us to install new digital signature algorithms securely long after anchoring.

SGX's `egetkey` cannot satisfy our ideal functionality. Lemma 1 says of IF that every successful icheck for P_s is preceded by P_s executing an iattest. If P_s receives a key k via `egetkey` and broadcasts it, then the adversary can create MACs that will pass `ckattest` without P_s executing `attestloc`. This pattern distinguishes the SGX-style mechanism from IF.

Nor can we provide a "logging enclave" based on `egetkey`, i.e. an enclave P_ℓ providing reliable logging functionality for other enclaves P_s. The obstacle is that P_ℓ must know P_s has taken the `attestloc`-like action to request the log record. If P_s discloses its `egetkey` key, or any key used to authenticate log requests to P_ℓ, then P_ℓ will emit the log record when other parties forge log requests. This again distinguishes the SGX-style logging enclave from IF.

In practice, there is often pressure to act on data in place, rather than copying it into local memory first. This has been the source of significant TrustZone attacks, exacerbated because the TrustZone "Secure World" has memory privileges [33]. Designers of systems based on CAIF will need care to handle these pressures safely.

B. Ideal functionality methods

§ VI shows that CAIF, when implemented with strong cryptography, is indistinguishable from the ideal functionality IF. Hence our protocol modeling also uses 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 [20], 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.

C. Protocol analysis

Analyzing security protocols has been a major undertaking since the late 1970s [35]; Dolev and Yao suggested cryptographic messages be regarded as forming a free algebra and using symbolic techniques [16]. A variety of formal approaches follow them, e.g. [9], [31], [42]. 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], [32], [4]. Our Thm. 1 provides some support for the soundness of our symbolic protocol analysis.

Our protocols read and write state records. State raises distinct problems from messages. These could be addressed in Tamarin, whose multiset rewriting model is fundamentally state-based [30]. By contrast, CPSA offers state in a primarily message-based formalism [22], [41], [23]; for its current treatment of state, see its manual [29, Ch. 8]. Squirrel interestingly expresses state in a computationally sound way [3].

CPSA has, helpfully, two modes. As a *model finder* it computes the set of essentially different, minimal executions [21]. 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 [43].

XI. Conclusion

CAIF's minimal hardware ensures the local service-to-service *provenance* of data, and its *protection* 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 a sequence of protocols to run atop CAIF. They start with an initial secure anchoring, a ceremony in a protected space, to establish a secret k_s shared with an authority. This key and its derivatives, protected by `protfor`, yield 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 one core need for secure reprogramming: it *authorizes* new programs for remote interactions. A second need is a way to *deauthorize* old programs, blocking rollback attacks, in which the adversary benefits by interacting with a deprecated version. Thus, secure reprogramming also needs irreversible changes, either to prevent the old programs from running, or at least to block access to the keys that previously authenticated them. Although this appears to require little more than monotonic counters and constraints on which services can advance them, it 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] Guoxing Chen and Yinqian Zhang. MAGE: mutual attestation for a group of enclaves without trusted third parties. In *USENIX Security Symposium*, pages 4095–4110. USENIX Association, 2022.

[13] 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.

[14] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[15] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[16] Daniel Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[17] 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.

[18] 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.

[19] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.

[20] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, aug 1986.

[21] 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.

[22] Joshua D. Guttman. State and progress in strand spaces: Proving fair exchange. *Journal of Automated Reasoning*, 48(2):159–195, 2012.

[23] 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.

[24] 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.

[25] Intel trust domain extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf, Februrary 2022. ID 690419.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Antonio Muñoz, Ruben Rios, Rodrigo Román, and Javier López. A survey on the (in)security of trusted execution environments. *Computers & Security*, 129:103180, 2023.

[34] 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.

[35] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *CACM*, 21(12):993–999, December 1978.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] John D. Ramsdell and Joshua D. Guttman. CPSA4: A cryptographic protocol shapes analyzer. The MITRE Corporation, 2023. https://github.com/mitre/cpsa.

[43] 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.

[44] John Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.

[45] 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.

[46] Victor Shoup. A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, 2001. https://eprint.iacr.org/2001/112.pdf.

Contents