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 Crypto-graphically 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.

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.
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 svc0 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 kar which can be used only by a particular service with code hash arh; § VIII-C sets up kar 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 Ps with v and the tag τ. Logging returns τ in response.

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

Any active process can use ichck to see if Ps has logged itself as an authority for v. However, only a service can execute iattest, since only services have a persistent identity Ps. Some function Flog of Ps and v determines τ. To implement iattest, one would use a MAC as Flog, 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 Ps is making the data v available just to one recipient service Pr:

iprotect has two parameters, the intended recipient service principal identity Pr and a pointer to a region of data with some contents v. When executed by a currently active service identity Ps, the escrow functionality selects a randomized handle η. It stores a record in the lookup table protstore, indexed by (η, Ps, Pr), pointing to the value v. We write this (η, Ps, Pr) → v.

The logging functionality returns τ in response.

iretrieve has two parameters, the expected source service principal identity Ps and a handle η. When executed by a currently active service identity Pr, the escrow functionality looks up the index (η, Ps, Pr) in the table protstore. If any entry (η, Ps, Pr) → v is present in protstore, that v is returned to Pr. Otherwise, it fails.

Lemmas 2–3 below depend on iprotect’s choice of handles η. We assume it samples η from a distribution D1,p,p′, for data of length j escrowed by principal Ps for recipient Pp 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 ≠ j′, the supports supp(Dj,p,p′) and supp(Dj′,p,p′) 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; (Ps, v, τ) for ichck; etc.

Definition 1: An event is a triple (command, principal, result) of: a command; the executing service principal that causes it (or ⊥ 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 ⟨⟨ci, Ps, ri⟩⟩i<ℓ of events in which each command ci can cause the result ri when executed by principal Ps in some state that can arise from the preceding events ⟨⟨cj, Pj, rj⟩⟩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 α = ⟨⟨ci, Ps, ri⟩⟩ be an IF behavior.

1. If, for i < j, c_i = iattest(v) and c_j = ichck(P_i, v, r_i), then r_j = true.

2. If c_j = ichck(p, v, τ) and r_j = true, then for some i < j, c_i = iattest(v), P_i = p, and r_i = τ.

This lemma is independent of how Flog chooses tags. Lemma 1 makes no claim about sequential order; check confirms only
Properties of IF: Protection. The analog of Lemma 1 holds for iprotect, using the re-sampling assumption (iii):

Lemma 2: Let \( \alpha = ((c_i, P_i, r_i)) \) be an IF behavior.
1. If, for \( i < j \), \( c_i = \text{iprotect}(P_j, v) \) and \( c_j = \text{iretrieve}(P_i, r_i) \), then \( r_j = v \).
2. If \( c_j = \text{iretrieve}(p, \eta) \) and \( r_j = v \), then for some \( i < j \), \( c_i = \text{iprotect}(P_j, v) \), \( P_i = p \), 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 \( \alpha \nu \) in the variable \( \nu \) results from a behavior \( \alpha \) by replacing one occurrence of a bitstring in a command or result of \( \alpha \) with the variable \( \nu \). If \( b \) is any bitstring, \( \alpha \nu[b/\nu] \) is the result of replacing the occurrence of \( \nu \) by \( b \). The latter may not be a behavior at all, since this \( b \) may be incompatible with other events in \( \alpha \).

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

Lemma 3: Let \( \alpha \nu = ((c_i, P_i, r_i)) \) be a schematic behavior, where \( \nu \) is a variable occurring in an iprotect instruction \( c_i = \text{iprotect}(P_i, \nu) \) executed by \( P_i \). Let \( \ell \) be a length of plaintexts for which the result \( r_i \) is possible. By assumption (ii), there is a unique such \( \ell \). Let \( D \) be any distribution with supp\( (D) \subseteq \{0,1\}^{\ell} \).

Suppose there is no subsequent \( c_j = \text{iretrieve}(P_i, r_i) \) with this input \( r_i \) and \( P_j = P_i \). For every \( b \in \text{supp}(D) \):
1. \( \alpha \nu[b/\nu] \) is a behavior;
2. the probability \( Pr[v_0 \leftarrow D; v_0 = b] = \text{Pr}_{\nu}[v_0/\alpha \nu]\) 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 \( F_{\text{log}} \) and a family of distributions \( D_{\ell, \nu, \nu} \); each instance of IF is of the form \( \text{IF}; F_{\text{log}}(\{D_{\ell, \nu, \nu}\}) \). 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 \( svh = \text{hash}(svAlg) \).

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 iattests a verifier record containing \( vk \) and \( svh \) and also 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 \( svh = \text{hash}(svAlg) \): it obtains a verifier record containing a verification key \( vk \) and its own hash \( svh \). If the record ichecks as logged by \( arh \), then \( svh \) awaits requests \((m, \sigma)\) from clients. If the signature verification succeeds for \((m, \sigma)\) with key \( vk \), then the service iattests a confirmation record containing yes, \( m \), and \( \sigma \). On failure, it 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 \((m, \sigma)\), 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 \( \sigma \) is received, and it 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 \((m, \sigma)\). 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 $h_{sr}$ 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 $sve_0$ logs $m_0$ that says, “I computed $v_0$ from $v_1$, which was logged by $sve_1$.” Local services can check the $m_0$ log entry, ensuring $sve_0$ logged it. If the $sve_0$ code does not log $m_0$ unless it checks $v_1$ and computes $v_0$, then $sve_1$ logged $v_0$. If $v_1$ is a message $m_1$ of similar form, we can follow the chain backward.

The inference requires knowing what $sve_0$’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 $sve_1$ 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 $\eta_1$.

**How to use CAIF’s local guarantees remotely.** Anchoring (§ VIII) and signature delegation (§ IX) yield authenticated messages from known services $sve$ on $d$. If $sve$ reports locally validated provenance chains on $d$, the peer learns the source of $sve$’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 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 \( \text{kdf}_h(x), \text{mac}_{\text{IS}}(k, v) \), and \( \text{enc}_{\text{IS}}(v, k) \) for the hardware key derivation function, MAC, and authenticated encryption. The corresponding hardware decryption is \( \text{dec}_{\text{IS}}(v, k) \).

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

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

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

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 *lattest* and *icheck* for asserting and checking provenance, and *iattest* 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 \( \text{kdf}_{\text{IS}} \), a Message Authentication Code \( \text{mac}_{\text{IS}} \), and an authenticated symmetric encryption \( \text{enc}_{\text{IS}} \) with the decryption \( \text{dec}_{\text{IS}} \). Thus, when \( c \) was not generated by an invocation of \( \text{enc}_{\text{IS}}(p, k) \), then \( \text{dec}_{\text{IS}}(c, k) \) 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
The encryption key \( k \) is computed via \( \text{kdf}_h \) as:

\[
k = \text{kdf}_h("\_\_f\_", IS, sh, sh_e).
\]

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

\[
e = \text{enc}_h(v, k)
\]

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

\( \text{retrvfm}(sh_e, e) \): Decrypts \( e \)—given an expected source’s service hash \( sh_e \) and a purported encryption \( e \)—using key \( k \) derived from \( IS \), the source’s hash \( sh_e \), 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 \( \text{kdf}_h \) as:

\[
k = \text{kdf}_h("\_\_f\_", IS, sh_e, sh_e).
\]

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

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

C. Auxiliary operations

We also need some auxiliary operations on services:

\( \text{create-service} \): 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.

\( \text{start-service} \): Given a runnable service, start it executing with access to the values of any additional parameters.

\( \text{yield} \): Stop executing the current service, retaining its state for future \( \text{start-services} \).

\( \text{exit-service} \): 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 \( (c_j, p_j, r_j) \) 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 Adv\(_{\text{prf}}(A, k, q)\) 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 Adv\(_{\text{mac}}(A, k, q)\) 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 Adv\(_{\text{ind-CCA2}}(A, k, q)\) measures \( A \)’s ability to distinguish encryptions of two different plaintexts, and the ciphertext unforgeability advantage Adv\(_{\text{eu-mac}}(A, k, q)\) 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\(_h\), (enc\(_h\), dec\(_h\)) used in the construction, as defined in § V-D.

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

In any state any query determines a distribution \( D \) 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 \( D \)-support for the previous state and the current query. More formally:

Definition 3: An oracle is a tuple \( O = (\Sigma, Q, I, R, \delta) \) such that \( I \subseteq \Sigma, \bot \in R, \) and \( \delta: \Sigma \times Q \rightarrow D(\Sigma \times R) \).

\( \Sigma \) is the set of states, \( I \) being the initial subset. \( Q \) is the space of possible queries. The function \( \delta \) is the probabilistic transition relation. An alternating finite sequence

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

(1)

is a trace of \( O \) 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. \( (\sigma_{j+1}, r_{j+1}) \in \text{supp}(\delta(\sigma_j, q_j)) \).

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

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

We use behavior and history interchangeably.

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

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

Where \( H = \langle (q_i, r_{i+1}) \rangle \) is a history, query(\( H \)) refers to the sequence \( \langle q_i \rangle \). We write \( v \in_S S \) when \( S \) is a sequence and \( v \) occurs in the \( i \)th position of \( S \).

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 protstore. The transition relation \( \delta \) is defined in § III-B using \( F_{\text{log}} \) and the family \( D_{l, p, p'} \). We identify the instruction attestloc with latestt, 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:

\[
\text{Adv}_{\text{att}}(A, C, k, q, t) = \Pr[(p, v, \tau); \mathcal{H}] = A_{q,t}^{C_k}(\cdot):
\]

\[
\exists i. \forall j < i. (\text{icheck}(p, v, \tau), p', \text{true}) \in_i \mathcal{H} \land
\]

\[
\text{(iattest}(v), p, \tau) \notin_j \mathcal{H}
\]

2. The **protection unforgeability advantage** of \(A\) is:

\[
\text{Adv}_{\text{pro}}(A, C, k, q, t) = \Pr[(p, v, \eta); \mathcal{H}] = A_{q,t}^{C_k}(\cdot):
\]

\[
\exists i. \forall j < i. (\text{iretrieve}(p, \eta), p', v) \in_i \mathcal{H} \land
\]

\[
\text{(iprotect}(p, \eta), p, \eta) \notin_j \mathcal{H}
\]

3. Let \(I^j(m, P, P_r)\) query \((\text{iprotect}(P_r, m), P_s)\) with result \(\eta\). The protection confidentiality advantage of \(A\) is

\[
\text{Adv}_{\text{pro-c}}(A, C, k, q, t) = |p_0 - p_1|\text{ where }p_0 = \Pr[(m_0, m_1, P_s, P_r, \alpha) \leftarrow A_{q,t}^{C_k}(\cdot),\eta \leftarrow I^{C_k}(m_0, P_s, P_r);\]

\[
(x; \mathcal{H}) = A_{q,t}^{C_k}(\cdot, \eta) = 1 \land |m_0| = |m_1| \land
\]

\[
\text{(iretrieve}(P_s, \eta), P_r) \notin \text{query}(\mathcal{H})]
\]

**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 \(\Gamma_2\) with additive computational overhead, such that for any \(A\),

\[
\text{Adv}_{\text{att}}(A, \text{CAIF}, k, q, t) \leq q\text{Adv}_{\text{eu-mac}}(\Gamma_{1,q}(A), k, q) + \text{Adv}_{\text{prf}}(\Gamma_{2}(A), k, q + 1)
\]

**Proof:** See [17] for proofs.

**Lemma 5** (Protection unforgeability): There are reductions \(\Gamma_{3,q}\) and \(\Gamma_4\) with additive computational overhead \(t_3(k, q)\) and \(t_4(k, q)\) respectively, such that for any \(A\),

\[
\text{Adv}_{\text{pro}}(A, \text{CAIF}, k, q, t) \leq q\text{Adv}_{\text{eu-enc}}(\Gamma_{3,q}(A), E, k, q, t + t_3(k, q)) + \text{Adv}_{\text{prf}}(\Gamma_{4}(A), \text{kdf}_h, k, q + 1, t + t_4(k, q))
\]

**C. Proving secure implementation**

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

\[
F_{\text{log}}(v, P) = \text{mac}_h(sk, v)\text{ where }sk = \text{kdf}_h(\text{"at"}, IS, P);
\]

\(D_{t,P_r,P_s}\) is the distribution generated by \(\text{enc}_h(0^t, sk)\) where \(sk = \text{kdf}_h(\text{"p"}, IS, P, P_s).

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

\[
\]

There are reductions \(\Gamma_{7,q}, \Gamma_8, \Gamma_{9,q}\), 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 \(\Gamma_{1,q}, \Gamma_2, \Gamma_{3,q}\), and \(\Gamma_4\) of Lemmas 4–5, s.t. for all \(q < W\),

\[
\text{Adv}_{\text{imp}}(A, \text{CAIF}, k, q, t) \leq q^2\text{Adv}_{\text{indCCA2}}(\Gamma_{7,q}(A), k, q) + \text{Adv}_{\text{prf}}(\Gamma_{10}(A), k, q)
\]

**Proof sketch:** The proof operates as a hybrid argument regarding four probabilities, namely the probabilities that \(A_{\text{IF}}(\cdot)\) outputs 1 for various oracles \(O_i\) for \(0 \leq i \leq 3\). The oracles are defined as follows:

\(O_0 = \text{IF}\).

\(O_1\) implements IF, except that for iprotect queries, we select the candidate values \(c\) by encrypting the input value \(v\) rather than \(0^{|v|}\).

\(O_2\) implements \(O_1\) except that it uses \(\text{kdf}_h\) with an intrinsic secret \(IS\) instead of using the random function \(R_k\).

\(O_3 = C_k\).

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 \(\text{kdf}_h\), existentially unforgeable mac, collision-resistant code hashing, and IND-CCA2 enc, 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 identifierimid; imids, 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
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 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 re-running 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 $kdf(k_s, sh)$ 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 $k_{ar} = kdf(k_s, arh)$.

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 $(dk, dvk)$. Using the shared secret $kdf(k_s, suh)$ 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 zeroes 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 $(sk, vk)$. It emits a certificate-like binding $[\ldots imid, sh, vk, \ldots] dk$ 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 in whatever unexpected patterns would benefit the adversary:

- **wc-protect** causes an iprotect instruction with current service $s_h$, recipient $s_h$, 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 $s_h$ 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 $s_h$ is the active hash in a wildcat role, then $s_h$ 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
   
   $\langle \text{imid}, \text{anch}, dh, n, r, dh \rangle$ 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
   
   $k_s = \text{kdf}_h(r, \text{imid})$.

   and replies with $n$ to confirm completion.

4. It uses protfor to protect $k_s$ together with the identities $\langle dh, \text{anch} \rangle$ for service $dh$ exclusively into shared storage; it zeroes its unshared memory and exits.

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 $r = \text{kdf}_h(r_0, \text{imid})$, so $k_s = \text{kdf}_s(\text{kdf}_h(r_0, \text{imid}), \text{imid})$.

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-imal 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 trech is a trust chain, we write the effect of pushing a new hash $h$ to it as $h :: trech$, 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 $n$th 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 $trech = skdh :: anch :: ()$, 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 $trech'$;
- a payload $payld$ to pass to the service $tgth$ when it runs.

The distributor service checks that $trech' = trech$. If so, skdh derives the key $k = kdf_h(k_s, tgth)$. It protects an extended trust chain, the payload, and this key $k$ for $tgth$:

\[
dprotfor\ tgth\ (imid,\ (tgth::trech),\ payld,\ k)
\]

into shared storage, after which it zeroes its unshared memory and exits. On device $imid$, $k$ can only be obtained by the service $tgth$. Since the DA can compute $k = kdf_h(k_s, tgth)$, $k$ 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 $tgth$.

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 $(sk, vk)$;
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 $shub$.

The delegation set-up service acts as follows:

1. It obtains a certify request ultimately from the CA containing a fresh serial number for the resulting certificate;
2. It generates a signing key pair \((dk, dvk)\) for \(dsh\);
3. It transmits a proof-of-possession message using the signing key \(dk\) to the \(CA\) on an authenticated channel;
4. It receives a certificate; and
5. It protects \(dk\) together with additional information for the sole use of \(dsh\) 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 \(m_3 = \left[ m_0 \right]_{sk}\) 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 \((sk, vk)\), 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:

\(\left[ serial, imid, dsh, suh, trch, dvk \right]_{dvk}\)

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:

\(\left[ imid, dsh, suh, trch, serial, dvk \right]_{CA}\)

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

\(\left[ imid, sh, trch', n, vk \right]_{dvk}\).

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 \(\left[ imid, dsh, suh, trch, serial, dvk \right]_{CA}\) for the delegation verification key \(dvk\);
\(m_2\): a delegation certificate \(\left[ imid, sh, trch', n, vk \right]_{dvk}\) for the service \(sh\)’s key \(vk\); and
\(m_3\): a message \(\left[ m_0 \right]_{sk}\) in which \(m_0\) is signed by some \(sk\) forming a key pair \((sk, vk)\) with the verified \(vk\).

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 \(\left[ m_0 \right]_{sk}\) can be observed, subject to (i)–(iii).

In this scenario, \(\left[ m_0 \right]_{sk}\) 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.

D. 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 \(ksud = \text{kd}_h(k_a, suh)\); \(suh\) 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
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 trch1 and the derived key kstellar. The trust chain trch is the DA’s acceptable chain of custody for kstellar, while trch1 contains its actual history, as analysis confirms. The set-up service does not proceed unless trch = trch1.

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 kstellar 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 ks 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 kdfh("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 kdfh("c", IS, src); the data destination, receives kdfh("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 Ps is preceded by Ps executing an iattest. If Ps receives a key k via egetkey and broadcasts it, then the adversary can create MACs that will pass ckattest without Ps 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 Ps providing reliable logging functionality for other enclaves Ps. The obstacle is that Ps must know Ps has taken the attestloc-like action to request the log record. If Ps discloses its egetkey key, or any key used to authenticate log requests to Ps, then Ps 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 crypto-
graphic 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


**CONTENTS**

I Introduction 1

II Current challenges 2

II-A Background challenges 2

II-B A CAIF application: Satellite reprogramming 2

III An Ideal Functionality for CAIF 3

III-A Main elements 3

III-B Behavioral lemmas about IF 3

IV Using the CAIF functionality 4

IV-A Satellite reprogramming via CAIF 4

IV-B Techniques for effective CAIF use 5

V CAIF Devices 6

V-A CAIF control over services 6

V-B CAIF instructions 6

V-C Auxiliary operations 7

V-D CAIF devices 7

VI CAIF securely implements IF 8

VI-A Defining advantages 8

VI-B Two lemmas 9

VI-C Proving secure implementation 9

VII Assured Remote Execution Strategy 9

VII-A Achieving Assured Remote Execution 9

VII-B Compliant roles and adversary roles 10

VIII Anchoring a CAIF Device 11

VIII-A Analyzing symmetric anchoring 11

VIII-B Trust chains 11

VIII-C The symmetric key distributor service 12

VIII-D Analyzing symmetric key distribution 12

IX Delegating Signing Keys 12

IX-A Assumptions and security goal for delegation 13

IX-B Message forms 13

IX-C Delegation analysis 13

IX-D Adapting delegation to the anchor 13

X Related Work 14

X-A Trusted Execution Environments 14

X-B Ideal functionality methods 14

X-C Protocol analysis 14

XI Conclusion 15

References 15