Understanding Attestation: Analyzing Protocols that use Quotes

Joshua D. Guttman and John D. Ramsdell

The MITRE Corporation

Abstract. *Attestation* protocols use digital signatures and other cryptographic values to convey evidence of hardware state, program code, and associated keys. They require hardware support such as Trusted Execution Environments or Trusted Platform Modules. Conclusions about attestations thus require reasoning about protocols, relevant hardware services, and possible behaviors of programs jointly.
This paper presents a mechanized approach to modeling these properties. Cryptographic Protocol Shapes Analyzer CPSA now combines protocol analysis with *axioms* or *rules*, allowing formalizing hardware and software conclusions.
We use CPSA to model aspects of Intel's SGX mechanism. We model underlying manufacturer-provided protocols, and build modular layers of attestation above this basis. User-level protocols can make trust decisions based on the results of attestation.

1 Introduction

Cryptographic protocols are often designed for use with particular software and hardware. How can we craft the mechanisms so that they jointly achieve certain overall security goals? In achieving their goals, the protocols may rely on specific assumptions about the remaining components' behaviors. These assumptions or *axioms* yield security-relevant specifications for the remaining components. They focus the design and validation processes for the components, and allow us to decide whether to use existing components.

This codesign process for protocols and other mechanisms requires protocol analysis to explore the executions that satisfy the axioms for the other components' expected behaviors. In this paper, we use the CPSA protocol analysis tool [38], which the developers have enriched with the ability to apply axioms or, as they are also called, *rules* [37]. The axioms it allows are implications, or more specifically universally quantified implications. They formalize the behavioral assumptions on the software and hardware context. CPSA then infers consequences about what can happen in different scenarios; these consequences are instances of axiom conclusions for which the hypotheses are satisfied.

CPSA implements *enrich-by-need* protocol analysis. The analyst selects a scenario of interest—perhaps, that one participant has had a successful local run, a couple of keys are uncompromised, and a nonce has been successfully chosen to

be fresh—after which CPSA displays all of the minimal, essentially different executions compatible with it [24,33]. CPSA can also "read off" a strongest security goal (e.g. authentication or confidentiality) that holds for that scenario [40].

The CPSA authors enriched CPSA to apply axioms in addition to the protocol-driven steps of the old CPSA, which we will call OLD CPSA. CPSA with axioms checks if a protocol is using its context correctly. The analysis codifies what matters about this context, focusing attention on whether the components satisfy the axioms for further formal or empirical investigations.

Other rigorous protocol analysis tools (e.g. Tamarin [42,34] and ProVerif [6,7]) can doubtless support variants of our method, which seems to us to increase its value. Adapting the method requires expressing the axioms so that the new tool can incorporate their protocol-relevant consequences into its reasoning.

Attesting to Trusted Execution Environments. We illustrate how to design protocols in system context by examining *attestation* for *trusted execution environments* or TEEs. A trusted execution environment is a software entity—either a thread with some memory or a virtual machine—that the processor promises to protect. Specifically, the processor will encrypt the TEE's memory before evicting it, and decrypt it only to return it to the same TEE.

An *attestation* for a TEE is a digital signature or Message Authentication Code that asserts that a TEE E is under the control of particular code C, and may associate other data D with E and C. Attestations, also called *quotes*, require support from the processor that must guarantee the TEE.

As we use TEEs, the other data D always includes a public key K, either a signature verification key or a public encryption key. The corresponding signing key or private decryption key K^{-1} should be under the control of the TEE, which inserts K into D. Thus, any remote entity that obtains an attestation for $E, C, K, \ldots$ can use K to create secure channels to E. Messages over these channels are entrusted to the code C. It may then follow that E uses K only in accordance with a protocol, if it is faithfully implemented in the code C.

TEEs are available as threads with protected memory within user-level processes on recent Intel processors. These so-called *enclaves* use the instruction set extension Software Guard Extensions (SGX) [28]. TEEs, as virtual machines, are available on AMD processors (Secure Encrypted Virtualization [30]). Other manufacturers may offer TEEs; academic work such as Sancus [36], for embedded systems, also provides TEEs. Our methods are applicable well beyond SGX, which currently has weaknesses [12,10,46].

Case study. Our case study illustrates building substantial mechanisms in layers that use protocol analysis and assumptions about hardware and software.

At the lowest level, we represent the mechanisms for SGX attestation, in which Intel has imposed some obstacles, such as online interaction with an Intel attestation server.[1] We identify three axioms that jointly characterize what the

[1] Intel has recently released an alternative to the attestation server infrastructure [29]. We will examine it with our methods subsequently.

hardware is intended to ensure, and how the provisioning of a signature key to the processor provides a supply-chain guarantee.

On top of the lowest layer, we identify a protocol and axioms that allow an enterprise to root subsequent attestations in its own key management architecture, after benefiting from a first supply-chain confirmation from Intel. Two axioms characterize the trust requirements on the key management architecture. Two others specify behavioral requirements on enclave code that implements this protocol, which we call the *crowbar*.

Finally, we illustrate how to use the attestations from the crowbar layer to draw conclusions about a user-layer protocol.

Contributions. We demonstrate how to combine axiomatic specifications and protocol analysis to design protocols targeted to hardware and software contexts. A benefit of the method is that it provides simple descriptions of what the protocol requires from these contexts.

The axioms we use fall into simple patterns that appear to be reusable for many attestation mechanisms.

Hardware axioms codify the relevant behavioral consequences of the manufacturer's claims about the processor.

Trust axioms formalize the decisions and practices of an organization about creating certificates and using the keys certified in them.

Attestation axioms apply only when a TEE is executing known code C; they express a behavioral specification for that code C, such as how it will handle its private keys. Static analysis and empirical testing, such as for side channels, can justify these axioms, or refute them [35]. A benefit of our approach is that it furnishes precise goals to prove or refute in these ways.

While other sorts of axioms also fit our formalism, these three types were central in applying the formalism to attestation and TEEs. They mechanize some of the reasoning in previous work on attestation for secure systems design, e.g. [15].

The axiomatic inferences fit smoothly into CPSA's existing structure. Indeed, CPSA is an excellent interactive tool for determining the relevant axioms. We derived the ones in this paper by observing what CPSA could *not* establish. We then introduced successive axioms that would provide it with information it needs, respecting the apparent intentions of the hardware and system designers.

Our work is a descendent of *authentication logics* [31,1], which were special-purpose logics for system designers to determine trust relations. Subsequent work showed how to use standard logics (Datalog in the case of [32]), and how to connect them with protocols [47,27,26,22]. We add a clear axiomatic structure for the combined analysis.

A *non-contribution* of this paper is any evidence that the axioms are true. Instead, we identify simple, relevant axioms that—*if* true—suffice to ensure that the application will meet its goals. To determine *whether* they are true in a particular instantiation calls for other—largely independent—methods, tuned to the claims of the hardware, trust, and attestation axioms. Our job is to focus attention on strong enough goals for the different components.

Structure of this paper. Section 2 presents our model of the SGX protocols for local (MAC-based) quotes, remote quotes using the EPID signature scheme, and online validation. Section 3 presents our crowbar for attestations based on standard digital signatures. Section A shows how an application level protocol can use SGX and the crowbar reliably.

More specifically, Sections 2.1, 3.1, and A.1 describe the protocol actions at the SGX, crowbar, and application protocol levels. Sections 2.2, 3.2, and A.2 enumerate the axioms at each successive level. A summary of CPSA appears in Section 2.3. Sections 2.4, 3.3, and A.3 present the analysis at successive layers, determining what the protocols can do subject to the axioms.

Overall patterns in these axioms are discussed in Section 4, with related work and conclusions in Sections 5–5.

The new CPSA is available [37]. Input and output files for our work are available at URL `https://web.cs.wpi.edu/~guttman/pubs/understanding_attestation_example/`.

Notation. We write:

$\#(m)$ for the result of a hash function applied to m; and

$\mathsf{mac}(m, K)$ for a keyed hash or Message Authentication Code in which K is the key and m is the value being authenticated;

pmk for the MAC key on a processor, regarding *pmk* as naming the processor.

$\{|m|\}_K$ for an encryption of m with K, either a symmetric or an asymmetric encryption, depending on the type of K.

$[\![m]\!]_K$ is a digital signature prepared using K;

$[\![m]\!]^e_K$ is a digital signature using Intel's EPID algorithm.

$\mathsf{tag}\ m_0$ is the contents m tagged with the distinctive bitstring tag.

(K, K^{-1}) is a keypair for an asymmetric algorithm, with $(K^{-1})^{-1} = K$.

$\mathsf{sk}(A)$ is the principal A's private signing key, and

$\mathsf{vk}(A)$ is the public verification key other principals use to check them.

$\mathsf{pk}(A)$ is a public encryption key to prepare messages for A, and

$\mathsf{dk}(A)$ is the corresponding private decryption key.

Thus, $\mathsf{sk}(A)^{-1} = \mathsf{vk}(A)$ and $\mathsf{dk}(A)^{-1} = \mathsf{pk}(A)$.

Non-compromised keys. We *do not* build into our notation that $K = \mathsf{sk}(A)$ or $\mathsf{dk}(A)$ is really uncompromised, which we instead express by writing $\mathtt{Non}(K)$.

The content of $\mathtt{Non}(K)$ has two parts. The first is that no entity other than the intended one(s) possesses and can use the key K. This requires hardware and software to cooperate so a malicious adversary does not obtain its value.

The second part is that the intended entity uses it only in the ways that the protocol dictates. It is not used to sign/MAC/decrypt messages in any other situation. Thus, when the intended entity is an enclave E under the control of code C, then $\mathtt{Non}(K)$ induces a software requirement, namely to ensure that the code C uses the key only to prepare messages that the protocol dictates should be sent, and only subject to the control flow the protocol dictates.

This second aspect of Non justifies protocol analysis in taking cases based on the protocol definition when a key is known or assumed to be non-compromised.

A brief introduction to strands. A *strand* is a finite sequence of message transmission and reception events, which we call *nodes*. Some strands, called *regular* strands, represent the compliant behavior of a single principal in a single local protocol session. Other strands represent actions of an *adversary*, who may control the network and may carry out cryptographic operations using keys that are public or have become compromised. An *execution* (or *bundle*) involves any number of regular strands and adversary strands, with the proviso that any message that is received must previously have been sent.

A *protocol* Π is a finite set of strands called the *roles* of the protocol, together with additional assumptions we discuss later. The roles $\rho \in \Pi$ contain *parameters*, and the *instances* of ρ are the strands that result from ρ by plugging in values for the parameters. The *sort* of a parameter restricts the values that may be inserted in place of it. This set of instances—obtained from Π's roles by plugging in values for parameters—defines the *regular strands* of Π.

Figures 1, 3, and 5 show examples of roles. We write roles and other strands either vertically or horizontally with double arrows $\bullet \Rightarrow \bullet$ connecting successive nodes. Single arrows $\bullet\ \to\ m$ and $\bullet\ \leftarrow\ m$ indicate that message m is being transmitted or received at the node (resp.).

In an execution, some strands might contain only an initial segment of the nodes of a role. For instance, at a particular time, the reception in a local run of the *local-quote* role (Fig. 1) may have occurred, but with as yet no response. Then we say that this strand has *height* 1, rather than the height 2 it would have if the next step had occurred. For more information on strands as a basis for protocol analysis, see [24].

2 Attestation in SGX

Intel's SGX attestation mechanism involves four elements.

First, a *local quote* about a *subject enclave* σ can be verified by a *target enclave* τ resident on the same processor. The local quote is a Message Authentication Code (MAC) prepared with a secret $f(pmk, \tau)$ depending on τ plus a unique secret *pmk* permanently protected within each processor (the Master Derivation Key, in SGX-speak). This MAC covers the *Enclave Record* (ER) for the subject enclave σ on this processor. The ER includes a hash of some code controlling the enclave's behavior together with other components. The subject enclave σ creates a local quote by the EREPORT instruction.

The target enclave τ checks a local quote using the instruction EGETKEY to obtain the MAC key $f(pmk, \tau)$, after which it recomputes the MAC value itself. The target enclave τ must be resident on the same processor, because *pmk* is an argument in computing the key $f(pmk, \tau)$. Since τ is an argument, a misbehaving τ cannot use this to forge local quotes targeted at a compliant τ'. The enclave τ' will always be given a key $f(pmk, \tau')$, which with overwhelming probability will not validate a forged MAC made with $f(pmk, \tau)$.

Second, to obtain attestations for entities on other devices, a *remote quote* is required. Remote quotes are created by a particular enclave, the *quoting enclave*

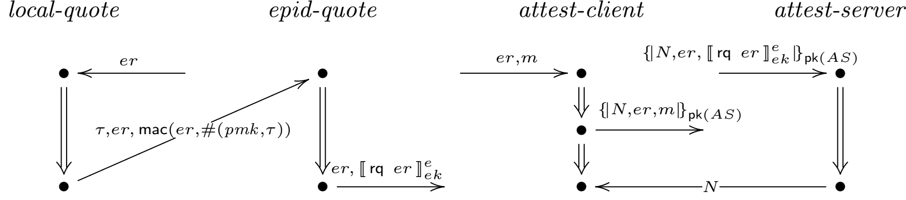


Fig. 1. SGX core roles

τ_q. It receives a claimed ER and a local quote. It checks the local quote against the claimed ER, using EGETKEY. On success, it generates a digital signature on ER using the group signature scheme EPID [11].

Third, Intel's *attestation server* validates remote quotes. A client connects via TLS, provides a claimed digital signature and ER, and receives an answer within the TLS connection. The attestation server vouches that some signing key provisioned by Intel created the digital signature on ER. The EPID group signature scheme prevents Intel from knowing which processor it was; the quoting enclaves they provision generates valid, but indistinguishable, EPID signatures.

Fourth, the *attestation client* queries the server over TLS.

We eliminate TLS's complexities, replacing it with a simple confirmation via public key encryption. This does not affect anything that matters to attestation. Any version of TLS that ensures integrity will lead to the same conclusions.

2.1 The Core SGX Protocol

The four roles of the manufacturer's mechanisms are shown in Fig. 1. The local-quote role does not run on every value er, but only on values that are in fact the enclave record of some enclave executing on the processor with secret pmk. In the EPID-quote role, the quoting enclave makes sure that its initial input has the form shown by executing EGETKEY on pmk. In the attestation-server role, the server receives a message encrypted with its public encryption key. Inside that message is a nonce N, which it will release just in case the remaining components er, $[\![\mathsf{rq}\ \ er]\!]^e_{ek}$ form a valid digital signature on er, formed using an EPID key ek generated in a protocol with the manufacturer as processors are prepared [11]. It thus provides a supply chain guarantee that the processor is genuine.

The attestation client's role corresponds, except that the client cannot directly determine that its input is of the form er, $[\![\mathsf{rq}\ \ er]\!]^e_{ek}$; it needs the attestation server precisely for this. Since the client cannot verify an EPID quote $[\![\mathsf{rq}\ \ er]\!]^e_{ek}$, the client may possibly submit any message m. If the strand successfully receives N, then in fact $m = [\![\mathsf{rq}\ \ er]\!]^e_{ek}$ for some EPID key ek. The attestation client chooses N randomly.

2.2 Rules for the SGX Protocol: Attestation and Trust

The analysis of the manufacturer's protocol relies on three rules. Each one codifies what follows when a role in Fig. 1 occurs. To express our rules, we need predicates that say when a strand is an instance of these rules, and to at least what height (number of steps). When a strand z engages in at least the first i transmissions and receptions of a role ρ, we write:

$\texttt{LocQt}(z, i)$ if ρ is the *local-quote* role;
$\texttt{EpidQt}(z, i)$ if ρ is the *epid-quote* role; and
$\texttt{AttServ}(z, i)$ if ρ is the *attestation server* role;

To refer to the values selected for role parameters, we write:

$\texttt{LocQtER}(z, er)$ if er is the enclave record value for *local-quote* instance z;
$\texttt{LocQtPr}(z, pmk)$ if pmk is the processor secret;
$\texttt{EpidQtKey}(z, K_{epid})$ if K_{epid} is the signing EPID key of *epid-quote* instance z;
$\texttt{EpidQtProc}(z, pmk)$ if z runs on the processor with secret pmk; and
$\texttt{ASQtKey}(z, K_{epid})$ if *attestation server* run z validates a quote signed with K_{epid}.

We use the special-purpose predicates $\texttt{EnclCodeKey}(\ldots)$ and $\texttt{ManMadeEpid}(\ldots)$. Although we give English-language descriptions for them, they are (formally) uninterpreted predicate symbols. Their significance comes from how the rules allow us to infer them, or infer further consequences from them.

Content of the rules. Starting with the *local-quote* role, when it executes on a valid SGX processor, what we can infer is that there is an SGX-protected enclave with the given enclave record. We will regard an enclave record as a sequence that starts with the enclave *id* number, the hash of its controlling code, and a public key, and may contain other entries subsequently. Writing :: for the list-construction operation, we thus have

$$er = eid :: ch :: k :: rest.$$

We refer to a processor by its processor secret pmk; even though no one knows this value, we can reason about whether $pmk = pmk'$, etc. Thus, a run of the *local-quote* role on a processor with non-compromised pmk implies that that there is an enclave characterized by the parameters eid, ch, k, pmk, which we will write $\texttt{EnclCodeKey}(eid, ch, k, pmk)$. Thus:

Rule 1 Quote guarantees enclave

$$\begin{aligned} &\forall z\colon \textsc{strd},\ eid, ch, rest\colon \textsc{mesg},\ k : \textsc{akey},\ pmk\colon \textsc{skey}\ .\ \texttt{LocQt}(z, 2) \wedge \\ &\quad \texttt{LocQtER}(z, eid :: ch :: k :: rest)\ \wedge\ \texttt{LocQtPr}(z, pmk) \wedge \texttt{Non}(pmk) \\ &\quad \Longrightarrow\quad \texttt{EnclCodeKey}(eid, ch, k, pmk). \end{aligned}$$

This straightforwardly states what a compliant processor's local quoting is supposed to tell us: It accurately reports an enclave running on that processor.

Second, when the Attestation Server completes a run, what must hold? It has ascertained that the purported EPID signature was in fact genuine, and

was produced using a key K_{epid} generated interactively with the manufacturer's EPID master secret. It can also vouch that the enclave mechanism can preserve the secrecy of K_{epid} within the EPID quoting enclave.[2] Hence:

Rule 2 AS says EPID key is manufacturer-made and non-compromised

$$\forall z\colon \textsc{strd},\ K_{epid} : \textsc{akey}\,.\ \ \texttt{AttServ}(z, 2) \wedge \texttt{ASQtKey}(z, K_{epid}) \\ \implies\ \ \texttt{ManMadeEpid}(K_{epid}) \wedge \texttt{Non}(K_{epid}).$$

The conclusion $\texttt{Non}(K_{epid})$ feeds back into the protocol analysis, since a non-compromised key often requires compliant local sessions to have occurred. The conclusion $\texttt{ManMadeEpid}(K_{epid})$ will also be used as a premise in the next rule.

The third rule offers us a conclusion when the *epid-quote* role executes a complete strand z with a valid EPID key. This is a supply chain property. It ensures that the processor is in fact an Intel-manufactured processor, which also generated an uncompromised processor secret *pmk*. Moreover, the processor is capable of preserving the secrecy of *pmk* and ensuring that it is used only in accordance with the roles shown in Fig. 1. Thus, the conclusion is simply $\texttt{Non}(pmk)$, stating that *pmk* is non-compromised, which again enables further protocol analysis conclusions.

Rule 3 Manufacturer-made EPID on non-compromised processor

$$\forall z\colon \textsc{strd},\ K_{epid} : \textsc{akey},\ pmk\colon \textsc{skey}\,.\ \ \texttt{EpidQt}(z, 2) \wedge \\ \texttt{EpidQtKey}(z, K_{epid})\ \wedge \texttt{EpidQtProc}(z, pmk) \wedge \texttt{ManMadeEpid}(K_{epid}) \\ \implies\ \ \texttt{Non}(pmk).$$

The manufacturer-made EPID key can be found only on a genuine processor, hence with a non-compromised processor secret (modulo [10,12,46]).

2.3 Protocol analysis with CPSA

Suppose that an *attestation client* has a run, following its role defined in the lower right of Fig. 1. We assume that it queries an attestation server AS with $\texttt{Non}(\mathsf{dk}(AS))$, and uses a fresh, unguessable nonce N. We also assume the purported enclave record to be of the form $er = eid :: ch :: k :: rest$. What else must then have happened, given the protocol of Fig. 1?

What CPSA does. A CPSA analysis starts with a scenario, such as we have just mentioned, in which some protocol activity is assumed to have occurred, which in this case is a regular *attestation client* strand. Moreover, additional facts may be included, which in our example are $\texttt{Non}(\mathsf{dk}(AS))$ and $\texttt{Unique}(N)$. The latter asserts that N was freshly generated and unguessable ("uniquely originating").

CPSA's job is then to find all minimal, essentially different executions that enrich the initial scenario; see [24] for much more detail. To find them, CPSA

[2] Since an out-of-order execution attack falsifies this claim [12], the current SGX does not satisfy our axioms. Cf. [10,46].

takes a succession of steps, exploring progressively more detailed scenarios—often with additional regular strands—until it finds some that are sufficiently rich. "Sufficiently rich" means:

1. Whenever a regular strand receives a message, the adversary can supply that message, possibly using messages transmitted previously by regular strands. The adversary has the usual, Dolev-Yao derivations [19], starting with initial values not ruled out by assumptions such as $\texttt{Non}(\texttt{dk}(AS))$ and $\texttt{Unique}(N)$.
2. R is a rule, and η instantiates its variables to values, making the hypothesis of R true. Then η yields a true instantiation of the conclusion of R.

The original OLD CPSA uses the *authentication test* idea [18] to find a small set of possible enrichments that are relevant in any case when a message reception does not satisfy Clause 1. To explain these receptions, OLD CPSA considers how to add new regular strands, and how to add new hypotheses about compromised keys. There may be different possible explanations to consider, causing branching in our search. We have left this functionality unchanged.

When Clause 2 is not satisfied for an R and an instantiation η, we want to add information to make that instance of the conclusion true. When the conclusion is an equation $s = t$, this means identifying the values associated with $\eta(s)$ and $\eta(t)$. When the conclusion is a conjunction of facts $P_i(t_1, \ldots, t_k)$, then we will add their corresponding instances $P_i(\eta(t_1), \ldots, \eta(t_k))$ to our scenario.

Although these are the only conclusions we use in this paper, the approach accommodates additional types. For *existentially quantified* conclusions $\exists x \,.\, \phi$, we consider both the existing values as well as a new value as the witness for the quantified variable x. When the conclusion is a *disjunction* (i.e. a logical *or*), we consider each branch separately in our search.

The hypothesis of each rule R is always an atomic formula or conjunction of atomic formulas. The resulting rules are thus *geometric sequents*, i.e. universally quantified implications in which the hypotheses are conjunctions of atomic formulas, and the conclusions are built from atomic formulas by conjunction, disjunction, and existential quantification, $\wedge$, $\vee$, $\exists$. These are precisely the syntactic forms of formulas that are preserved by all homomorphisms [25].

Correctness for this procedure means that it must yield, on termination, scenarios that cover all possible executions that enrich the initial scenario. For the original OLD CPSA, see [24]; for enrichment with geometric sequents, see [41,20,37].

CPSA's input and output. For a given protocol and rules, CPSA's input is a scenario consisting of some assumed strands of regular participants, together with some assumptions such as $\texttt{Non}(\texttt{dk}(AS))$ and $\texttt{Unique}(N)$ or other facts (closed atomic formulas). The starting scenario and similar structures are called *skeletons*. CPSA performs its search by fixing counterexamples to clauses 1–2 above, or in the case of OLD CPSA, just clause 1.

If its search terminates, CPSA returns a set of skeletons representing all minimal, essentially different executions that enrich the initial skeleton. This may be the empty set, when the initial skeleton cannot occur; e.g. it hypothesizes some

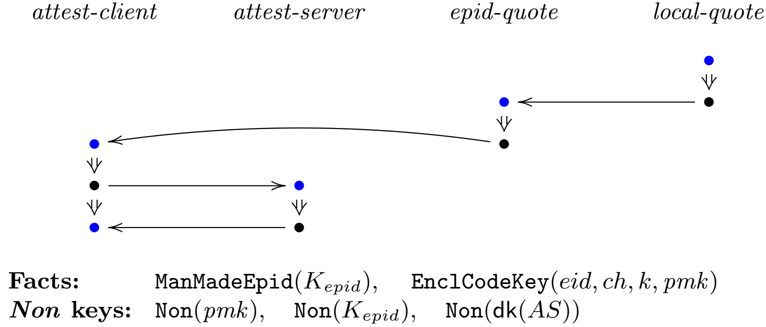


Fig. 2. Consequences of an attestation client success

security disclosure that CPSA shows cannot occur. Very often, this set is very small, containing only one or a few possibilities.

CPSA presents its results by diagrams, such as those in Figs. 2, 4, etc. Each diagram shows some strands, presented as vertical columns of transmissions and receptions, together with arrows summarizing ordering information among the events. Each skeleton also shows the parameter values of the different strands, and the other facts that hold in this skeleton.

2.4 Applying CPSA to the SGX protocols

In our case study, an *attestation client* has a run, following its role defined in Fig. 1. We assume it queries an attestation server AS such that $\texttt{Non}(\texttt{dk}(AS))$, and uses a fresh, unguessable nonce N. We also assume the purported enclave record to have the form $er = eid :: ch :: k :: rest$. What else must have happened, given the remainder of the protocol contained in Fig. 1?

We ask CPSA this question, subject to Rules 1–3. CPSA answers by computing the result shown in Fig. 2. The assumed attestation client run is shown as the leftmost column in Fig. 2. The keys K_{epid} and pmk are new, implicitly existentially quantified values. The client does not find out what they are, but knows they exist. CPSA computes this in three steps.

1. The first step introduces the attestation server run shown immediately to the right. CPSA infers this as a consequence of the protocol definition. Only an attestation server run can extract the nonce N from the encryption inside which the client transmits it. Rule 2 now applies to the new strand, introducing the facts $\texttt{ManMadeEpid}(K_{epid})$ and $\texttt{Non}(K_{epid})$.

2. Since CPSA now knows that the client run started by receiving a valid EPID-signed remote quote, CPSA explains it by a matching run of the *epid-quote* role. Its pmk parameter was previously unknown. By Rule 3, we infer $\texttt{Non}(pmk)$.

3. How was the local quote $\texttt{mac}(er, \#(pmk, \tau))$ generated? CPSA infers it can come only from a run of the *local-quote* role with matching parameters. Applying

Rule 1, it adds the fact that the enclave record describes an enclave running on *pmk*. This fact is expressed by $\texttt{EnclCodeKey}(eid, ch, k, pmk)$.

The analysis is now complete.

Omitting rules. Omitting Rule 1 does not change the diagram, but the fact $\texttt{EnclCodeKey}(eid, ch, k, pmk)$ is lost. We no longer know that there is an enclave controlled by the code with hash *ch* and public key *k* running on processor *pmk*.

Omitting Rule 3 omits this fact, as well as the (rightmost) *local-quote* strand. The key *pmk* is no longer known to be `Non`. Finally, omitting Rule 2 means that only the *attest-server* strand is available. Thus, each rule has a definite and predictable effect on how much of the analysis goes through.

Attestation consequence. We have now identified the exact consequences that follow from a successful attestation client run in favorable circumstances: A processor with *confirmed supply chain* generated a local quote for the enclave record *er*. On that same processor, a remote quote was created from the local quote. Finally, on that processor there is an *enclave* under control of the *known code ch* with associated key *k*.

By *favorable circumstances*, we mean first that *N* was freshly chosen, and that $\mathsf{dk}(AS)$ is used only in accordance with the protocol. More important, the favorable circumstances depend on the rules: The SGX hardware should ensure that a local quote on a processor with non-compromised *pmk* ensures a corresponding enclave (Rule 1); the attestation server succeeds only when the remote quote was generated with a properly provisioned, non-compromised EPID key (Rule 2); and the EPID key provisioning should ensure that a processor with an acceptable EPID key can keep its *pmk* non-compromised (Rule 3).

Making the three rules hold requires challenging—not yet fully achieved—processor engineering and cryptographic design and implementation [10,12,46]. However, the rules summarize the intended consequences of those tasks succinctly, transparently, and usefully for mechanized analysis.

3 Mechanical Advantage for Trust

We now show how to build a new attestation protocol on top of the mechanisms that the manufacturer provided. Partly, we do so to show how our analysis for the lowest layer of the composite protocol extends smoothly upward.

The other reason is practical. The manufacturer's protocol (Fig. 1) binds the processor at the end of a supply chain back to a device that generated an EPID key at the manufacturer's facility at the start of the supply chain. However, it verifies remote attestations via a public network, forcing disclosure to the manufacturer that an attestation is occurring.

This is often unacceptable. Devices may need to verify a remote attestation before connecting to a network, e.g. to determine if the network servers are trustworthy. Private networks may be shielded from the public internet, for instance at banks, and in industrial control systems and critical infrastructure. Other privacy and availability concerns may also apply.

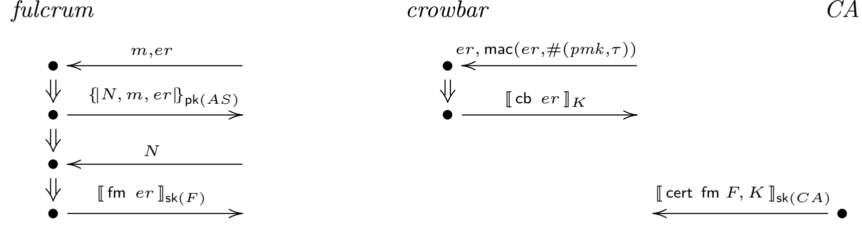


Fig. 3. The fulcrum, crowbar, and CA roles

3.1 The Fulcrum and the Crowbar

In Fig. 3 we offer a protocol to provide "mechanical advantage for trust." It allows attestations to be generated and verified privately, after a supply chain guarantee rooted in a single interaction with the manufacturer's attestation server.

The fulcrum is public and online, and interacts with the attestation server once per processor. It digitally signs an attestation about a long-term *crowbar* enclave.

The fulcrum F's signing key $\mathsf{sk}(F)$ uses a standard signature algorithm, e.g. the Eliptic Curve Digital Signature Algorithm (ECDSA). An organization standing up a fulcrum F also provides a certificate for its verification key $\mathsf{vk}(F)$, stating that signatures verified with $\mathsf{vk}(F)$ came from a valid fulcrum operated by this organization. F may be implemented within an SGX enclave to protect $\mathsf{sk}(F)$, but our protocols do not use attestations about this enclave.

When F receives a message m and an enclave record er, it queries an attestation server with $\{\!|N, er, m|\!\}_{\mathsf{pk}(AS)}$. Receiving N signals an affirmative result, validating $m = [\![\, \mathsf{rq}\ \ er \,]\!]^e_{\mathsf{sk}(b)}$ for some b. It then issues a tagged *fulcrum report* $[\![\, \mathsf{fm}\ \ er \,]\!]_{\mathsf{sk}(F)}$, where "fm" distinguishes fulcrum reports from other signatures.

A valid fulcrum F is trusted to choose a peer AS whose private decryption key matching $\mathsf{pk}(AS)$ is uncompromised and used only for the attestation server role. It is also trusted to choose fresh challenge nonces N.

The crowbar runs on a processor *pmk* that may be inaccessible from the public internet. It will digitally signing remote quotes for numerous enclaves running on *pmk*. Like the fulcrum, the crowbar uses a standard digital signature, which can be verified without further interaction.We call these *crowbar reports.*

When given a purported local quote targeted to it, a crowbar enclave uses EGETKEY to check it is a MAC generated using *pmk*. If so, it generates the crowbar report $[\![\, \mathsf{cb}\ \ er \,]\!]_{\mathsf{sk}(C)}$ tagged with cb as attestation.

When a processor *pmk* is set up, the fulcrum receives a remote quote about the crowbar—generated by the manufacturer's EPID quoting enclave on *pmk*—perhaps via carrier pigeon. If the fulcrum confirms this remote quote, it issues a fulcrum report, which will return to the private network in a similar way.

An application level client can use this fulcrum report and a crowbar report to justify a trust decision about the targets of the crowbar report. The client

also relies on a digital certificate from its organization's Certifying Authority to confirm a signing key for the fulcrum.

A certifying authority CA provides digital certificates for any trustworthy fulcrums. The organization's certificate for $K = \mathsf{vk}(F)$ is intended to assert that the organization believes that the matching signature key $\mathsf{sk}(F)$ is uncompromised, and will be used only for the fulcrum role.

3.2 Rules for the crowbar

The trust and attestation assumptions that govern the fulcrum and crowbar consist of four rules.

Rule 4 states that the CA behaves like a standard trust anchor, and vouches that its target's private key will be used only as permitted by the protocol.

Rule 5 asserts that a fulcrum run chooses a non-compromised attestation server (run by the manufacturer) as its peer. It also asserts that the fulcrum chooses its challenge nonce freshly.

Rule 6 asserts that any crowbar that runs acceptable code in an enclave successfully protects a non-compromised signing key.

Rule 7 states that the crowbar never migrates its signing key, i.e. that if there was ever an enclave such that $\texttt{EnclCodeKey}(eid, ch, k, pmk)$ with acceptable crowbar code ch, then there is never a run of the crowbar role using the same signing key on any different processor $pmk' \neq pmk$.

To express them, we will use predicates stating that a strand is an instance of a new role, with at least a given height, and that a strand has a particular instance for a parameter. When a strand z engages in the first i transmissions and receptions of a role ρ, we write:

$\texttt{Fulc}(z, i)$ if ρ is the *fulcrum* role;
$\texttt{CrBar}(z, i)$ if ρ is the *crowbar* role; and
$\texttt{Certifier}(z, i)$ if ρ is the *CA* role.

To refer to the values selected for role parameters, we write:

$\texttt{CertID}(z, f)$ if f is the name of the fulcrum certified in strand z, and
$\texttt{CertKey}(z, K)$ if K is the target fulcrum verification key;
$\texttt{FulcPeer}(z, as)$ if as is the peer attestation server chosen in strand z, and
$\texttt{FulcNonce}(z, n)$ if n is the challenge nonce;
$\texttt{CrBarPubK}(z, k)$ if K is the signing key used in crowbar strand z, and
$\texttt{CrBarPr}(z, pmk)$ if pmk is the processor executing z.

We will use a formally uninterpreted predicate $\texttt{CbCode}(ch)$. Informally, we use this to express the idea that the code C with hash ch is acceptable crowbar code, and uses signing key K only in accordance with the crowbar role. We write $\texttt{Unique}(v)$ for the assertion that the message value v is freshly chosen ("uniquely originating"). In particular, v is chosen only once by a compliant

principal, and will not be guessed by an adversary. CPSA interprets this in its protocol analysis.

Trust rules. Rules 4–5 are *trust rules*: One who accepts them does so out of trust that the organization behaves correctly. It should have audited the fulcrum code to ensure that it will protect its private signature key (Rule 4) and that it uses the PKI and random number generator correctly (Rule 5). This is merely procedural, as the party deciding to extend its trust receives no evidence through our protocols.

First, we regard the trust anchor as asserting that a signing key is non-compromised if it matches a certified verification key.

Rule 4 (CA trust anchor) $\forall z\colon \text{STRD},\ f\colon \text{NAME},\ K\colon \text{AKEY}\,.$

$$\texttt{Certifier}(z,1)\ \wedge \texttt{CertID}(z,f) \wedge \texttt{CertKey}(z,K) \wedge K = \mathsf{vk}(f)$$
$$\implies \quad \texttt{Non}(K^{-1}).$$

The hypothesis $K = \mathsf{vk}(f)$ restricts the applicability of the rule slightly. CPSA uses the notation $\mathsf{vk}(A), \mathsf{pk}(A)$ to associate the name A with its public signature verification and public encryption keys. $K = \mathsf{vk}(f)$ asserts that this notational convention is compatible with the CA's actions. In effect, we draw a conclusion about $\mathsf{vk}(A)$ only when the CA has actually issues a certificate for it.

This is a *trust rule*. Any principal that accepts the certificate is transferring some of its trust in the trust anchor CA to the certified fulcrum f.

The second rule requires a fulcrum to select a compliant attestation server. This means in effect selecting an attestation server with a non-compromised private decryption key $\mathsf{dk}(as)$. The fulcrum must also select a fresh nonce to send as a challenge. These two conditions are required to make the fulcrum useful; together, they assure that the analysis will discover a run of a compliant authentication server responding to the fulcrum's query.

Rule 5 (Fulcrum finds attestation server) $\forall z\colon \text{STRD},\ as\colon \text{NAME},\ n\colon \text{TEXT}\,.$

$$\texttt{Fulc}(z,4) \wedge \texttt{FulcPeer}(z,as) \wedge \texttt{FulcNonce}(z,n)$$
$$\implies \quad \texttt{Non}(\mathsf{dk}(as)) \wedge \texttt{Unique}(n)$$

Before the CA certifies f and its key $\mathsf{vk}(f)$, someone needs to inspect the code in control of $\mathsf{sk}(f)$, to ensure that it is using the manufacturer's PKI properly to ensure that it reaches a valid as using $\mathsf{pk}(as)$. This rule expresses the trust that the organization has done so successfully.

The $\texttt{Unique}(n)$ conclusion says that the nonce n is chosen freshly, or *uniquely originating*. Again, someone must ascertain this before the certificate is issued, by examining the fulcrum code. They want to ascertain that it generates n with good randomness. Our assertion that n is uniquely originating may idealize a claim that holds with overwhelming probability.

Attestation rules. By contrast, the next two rules are *attestation rules*. The party making a trust decision obtains evidence about the code controlling the enclave it is evaluating. This evidence always derives ultimately from a run of the *local-quote* role. Thus, if a rule draws some conclusion, the deciding party can

always inspect the evidence—and the code that yields the hash ch—to determine whether this code will act only in accordance with our protocols.

In rule 6, the conclusion is that the crowbar's signing key is non-compromised. If a deciding party wants to check this, that party should consider whether it belongs to a keypair that is generated within the crowbar enclave, and whether the private signing part ever contributes to transmitted data except by being used for signature via a sound algorithm.

If C is code with $\#(C) = ch$, a static analysis of C may be able to ascertain this. Accepting the premise $\texttt{CbCode}(ch)$ summarizes the conclusion of examining C. Although we do not model any aspect of this appraisal, we codify the consequences of the decision.

Rule 6 (Crowbar attestation) $\quad \forall eid, ch\colon \text{MESG},\ k \in \text{AKEY},\ pmk\colon \text{SKEY}\,.$

$$\begin{aligned}&\texttt{EnclCodeKey}(eid, ch, k, pmk) \wedge \texttt{CbCode}(ch)\\ &\quad\Longrightarrow\ \texttt{Non}(k^{-1})\end{aligned}$$

The last rule states that the processor doesn't change. This says more than the private key being non-compromised. It would still be non-compromised if the original enclave transferred it, through a secure channel, to another enclave that would still use it only for crowbar functionality. One may determine, by inspecting code, that it will not engage in that behavior. Thus, an attestation can also give evidence that the processor on which the crowbar key is used will not change.

Rule 7 (Crowbar immobile)

$$\begin{aligned}&\forall z\colon \text{STRD},\ eid, ch\colon \text{MESG},\ k\colon \text{AKEY}, pmk, pmk'\colon \text{SKEY}\\ &\quad\texttt{EnclCodeKey}(eid, ch, k, pmk) \wedge \texttt{CbCode}(ch)\wedge\\ &\quad\texttt{CrBar}(z, 2) \wedge \texttt{CrBarPubK}(z, k) \wedge \texttt{CrBarPr}(z, pmk')\\ &\qquad\Longrightarrow\quad pmk = pmk'\end{aligned}$$

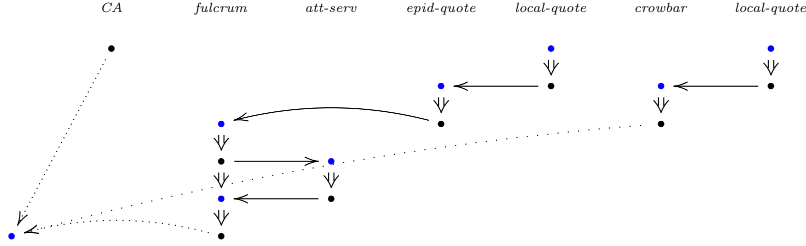
The rules—specifically, Rules 6–7—use $\texttt{CbCode}(\ldots)$ only as a hypothesis, and never in a conclusion, so the effect of deciding to assume instances of $\texttt{CbCode}(\ldots)$ is very clear: It determines which in which cases Rules 6–7 may apply. These two rules codify the significance of $\texttt{CbCode}(\ldots)$.

3.3 Protocol analysis: Crowbar level

Suppose that an application-level protocol acquires, for some CA with $\texttt{Non}(\mathsf{sk}(CA))$:

certificate $[\![\, \mathsf{cert\ fm}\ F,\ \mathsf{vk}(F)\,]\!]_{\mathsf{sk}(CA)}$;
fulcrum report $[\![\, \mathsf{fm}\ eid :: ch :: k :: rest\,]\!]_{\mathsf{sk}(F)}$ for the same F; and
crowbar report $[\![\, \mathsf{cb}\ er'\,]\!]_{k^{-1}}$ prepared with the matching signing key k^{-1}.

What then follows? If er' takes the form $eid' :: ch' :: k' :: rest'$, we would like to infer that the crowbar has attested a subject enclave eid', controlled by code ch', and controlling the key k'. The subject enclave should be guaranteed by the occurrence of a *local-quote* with this er', targeted to the crowbar enclave



Facts: $\texttt{ManMadeEpid}(K_{epid})$, $\texttt{EnclCodeKey}(eid, ch, k, pmk)$, $\texttt{EnclCodeKey}(eid', ch', k', pmk)$.

Fig. 4. Shape for a certificate, fulcrum report, and crowbar report

eid. The crowbar itself is the subject enclave of a *local-quote* on the same *pmk*, converted to a remote quote by an *epid-quote* strand.

One would expect this conclusion to hold only if $\texttt{CbCode}(ch)$, which is needed in Rule 6 to infer $\texttt{Non}(k^{-1})$. With the additional assumption that $er' \neq er$, CPSA derives the desired conclusion.

If $er' = er$, the *local-quote* strands may be identical. If $er \neq eid :: ch :: k :: rest$, they are certainly distinct. In the latter case, one local-quote attests to the enclave executing the crowbar, and its local quote is converted into an EPID quote by the EPID quoting enclave. The other local-quote is converted into a standard digital signature by a run of the crowbar role, as shown in Fig. 4.

The dotted arrows in Fig. 4 indicate a flow of information in which the value received is distinct from the value sent. In this case, the tuple of the three incoming values is received. CPSA reasons about inequality using the logical rule

Rule 8 $\forall m\colon \text{MESG} \,.\, \texttt{Neq}(m, m) \Longrightarrow \texttt{Falsehood}$.

The crowbar may now be used for application-level attestation. If properties of the target code ch' are known, these can be formulated in rules about enclaves running this target code. We show how to do this in Section A.

Omitting rules. Removing any of Rules 4–6 cause parts of the conclusion in Fig. 4 to be lost.

Omitting Rule 7 is more interesting. CPSA infers the presence of all but the rightmost strand of Fig. 4, the second instance of *local-quote*. Moreover, the instance of the crowbar has as its processor parameter pmk', some processor possibly $\neq pmk$, i.e. possibly distinct from the processor on which the *epid-quote* and its *local-quote* occurred. Hence, we do not know whether $\texttt{Non}(pmk')$ holds, and we cannot infer integrity for the MAC $\mathsf{mac}(er', \#(pmk', \tau))$. Possibly the adversary produced it, rather than a *local-quote* strand.

Without Rule 7, the key k may have migrated from the enclave *eid* to another processor. The process receiving k may not be an enclave, and the new processor

may not even be SGX-equipped. Hence the signature may have been applied to an enclave record er purportedly guaranteed by a compromised pmk'.

Including Rule 7 thus imposes a behavioral requirement on the crowbar: It must never migrate its key to another enclave or any remote process. Inspecting the code ch may confirm that it will never do so.

Using the crowbar from an application. Suppose that we have a protocol Π and two roles $\rho_1, \rho_2 \in \Pi$. A principal A executing role ρ_1 would like to authenticate a peer B executing role ρ_2. Suppose also that an analysis of Π shows that this holds *assuming* that a private key K_B is non-compromised.

We can then always use the crowbar to discharge this assumption:

1. We will embed the code C executing ρ_2 (apart from actual I/O) in an enclave. We introduce a predicate $\texttt{PeerCode}(ch)$ true of $ch = \#(C)$ only when C protects K_B and uses it only in accordance with ρ_2.
2. We codify this in an axiom

   Rule 9 (ρ_2 Attestation) $\forall eid, ch\colon \text{MESG},\ k \in \text{AKEY},\ pmk\colon \text{SKEY}\,.$

   $$\texttt{EnclCodeKey}(eid, ch, k_B^{-1}, pmk) \land \texttt{PeerCode}(ch) \implies \texttt{Non}(k_B)$$

3. We prepend a step before ρ_1 to collect a certificate, fulcrum report, and crowbar report.

In strands where the crowbar report's subject enclave satisfies $\texttt{PeerCode}(ch)$, A can infer $\texttt{Non}(k_B)$ and safely complete the run. In Appendix A, we illustrate this in detail in the case of a particular application level protocol.

4 Types of rules

We first categorize Rules 1–3 from Section 2. We then divide the rules we have used into three types: *hardware* rules, *trust* rules, and *attestation* rules.

Hardware rules. Rule 1 stipulates a *hardware* property, namely when the processor generates a local quote on er, there is an enclave with record er. Rule 3 is also, at least partly, a hardware requirement: a processor with a manufacturer-made EPID key protects pmk, and uses it only to generate and check local quotes. There is also a *trust* aspect: the manufacturer should not install a manufacturer-made key K_{epid} unless the processor can protect its secret pmk.

These rules define the hardware requirements. Naturally, the hardware's enclave support must also justify the code analysis leading to the attestation rules.

Trust rules. Rules 2, 4, and 5 are trust rules. Rule 2 expresses our trust that the manufacturer will operate a reliable Attestation Server, and it defines what we need from the AS, namely confirmation of the origin of K_{epid} and of its protection from compromise. However, there is no attestation here, since there is no evidence that particular code is in control of the AS. Hence there is no direct evidence the code will ensure the conclusions we care about.

Rule 4 expresses trust in our organization's CA. Specifically, CA must emit this type of certificate only when the private key $\mathsf{sk}(f)$ will be protected from disclosure and used only by code playing the role of *fulcrum*. Rule 5 expresses trust in our organization's fulcrum code, specifically its ability to use the PKI correctly to find the manufacturer's AS. Trust in the manufacturer is also needed, namely to protect $\mathsf{dk}(as)$ and allow only the proper AS code to use this key. Again, there is no attestation here, since no quote provides evidence that particular code is in control and will behave correctly.

Attestation rules. Rules 6, 7, and 9 are attestation rules. Each of them has a premise $\mathtt{EnclCodeKey}(eid, ch, k, pmk)$, so they apply only when other considerations have already established an enclave with code (hashing to) *ch*.

A rational process governs proposed enclave rules. One can analyze the behaviors of the known code. Does it randomly generate the keypair (k, k^{-1}) and installing k in the enclave record? Does it protect the private k^{-1}, using it only in secure cryptographic algorithms? What holds (empirically or by code analysis) about side channels? Does the code use its keys only for transmissions and receptions following the specific roles in which this key is expected to engage?

In attestation rules, we always know what code is in control, and we know that it executes within an enclave. Thus, we can use well-understood methods to ascertain whether the conclusion follows.

Uses of the rules. The rules are valuable. First, they provide simple *specifications* of the relevant components. The hardware rules make clear what we need from SGX. The trust rules provide guidelines for organizations' CAs, fulcrums, and—in the case of the manufacturer—the public attestation server. The attestation rules specify what behavior to permit from the attested code *ch*.

Hence, the rules provide *guidance to an implementer* about how to build the components correctly. If components already exist, they should provide advice to a formal analyst who would like to prove that these components will live up to their purpose within the mechanism.

They also help the *red team* that would like to find out how the mechanism can fail. It says which misbehaviors in the pieces would lead to failure of the mechanism. Also, *testing* gets improved focus from these succinct, intuitive rules.

Developing the rules. CPSA is an excellent assistant for developing rules. It gives quick interactive feedback when rules are too weak. This allows a designer to balance out the security goals she expects the system to achieve against the requirements she is willing to impose on the remaining components. CPSA's graphic output makes the effects of particular choices very clear. Its speed is very helpful; no individual run in the development of this paper took more than a second on a standard laptop.

Broader types of rules. Our rules have only a few forms. Some conclusions assert that a key K is non-compromised, i.e. $\mathtt{Non}(K)$. Rule 7 asserts an equality, $pmk = pmk'$. A few others assert facts $\mathtt{EnclCodeKey}(\ldots)$, $\mathtt{ManMadeEpid}(\ldots)$, and $\mathtt{Unique}(n)$.

They are particularly simple because our protocols are well-structured. We have used tags—i.e. the constants $\mathsf{rq}, \mathsf{fm}, \mathsf{cb}$, and $\mathsf{cert\ fm}$—to make them syntactically unmistakable. Thus, principals cannot misinterpret the purpose of a message [2]. This is good practice whenever the designer has control over the message formats. However, sometimes the formats have already been defined.

We also studied a variant of the SGX and crowbar protocols omitting all tags $\mathsf{rq}, \mathsf{fm}, \mathsf{cb}$, and $\mathsf{cert\ fm}$. Initially, CPSA reported many possible confusions among roles. Some of these may be eliminated by distinguishing cryptographic primitives. For instance, an EPID signature is generated by the EPID quote role, but will never be generated by the fulcrum role, and the attestation server will validate only an EPID signature. Additional rules can capture these ideas.

Moreover, the trust anchor key $\mathsf{sk}(CA)$ will never be used as a fulcrum key or a crowbar key, eliminating some instances of these roles.

An enclave with code for one of the roles will never engage in a different role. For instance, the code that implements the crowbar can never act in the *fulcrum* role. It lacks the logic to contact an Attestation Server with a properly formatted query, as a fulcrum does. This justifies a rule:

$$\begin{aligned}&\forall z\colon \textsc{strd},\ f\colon \textsc{name},\ K\colon \textsc{akey}\ .\ \mathtt{EnclCodeKey}(eid, ch, \mathsf{vk}(f), pmk) \wedge \\ &\quad \mathtt{CbCode}(ch)\ \wedge\ \mathtt{Fulc}(z,4)\ \wedge\ \mathtt{FulcSelf}(z,f) \\ &\qquad \Longrightarrow\quad \mathtt{Falsehood}.\end{aligned}$$

This is a natural requirement on the software implementing the crowbar.

We reproduces the same analysis we showed in Section 3 without tags, using a total of 21 rules. We consider the division of labor between protocol structure and rules that we presented in Sections 2–3 to be cleaner and more convincing than this version with twice as many rules. However, when protocol structure cannot be changed, rules of these broader kinds are quite usable.

5 Related work and Conclusion

We will highlight three areas of related work.

Security protocol analysis is a very well-developed field, with numerous sophisticated tools for trace properties (e.g. [6,17,21,34,38]), and some for determining indistinguishability properties also (e.g. [8,9,13,14]). In many cases, our work is compatible with other approaches rather than in competition with it. For instance, tamarin [34] has a notion of *restriction* used to restrict the traces of interest. It may also be possible to build similar conditions into ProVerif's resolution back end [6]. This increases the value of using rules to formalize the context in which protocols run: Multiple tools can shed light on the consequences.

Enrich-by-need is specific to CPSA, however. This is very useful in development, as CPSA provides a complete overview of the minimal, essentially different possibilities. This shows which (true) rules should be added to the specifications, or what strengthenings of the protocol are needed so that true rules can suffice.

Connecting security protocols to context has been less studied than one would expect. There are many cases where the protocol should inform the

application it serves of security-relevant events. For instance, the renegotiation attacks on TLS [39] arose because the protocol could not signal to the application level when the authenticated identity of the peer changed. As a consequence, an adversary can benefit by prepending an unauthenticated flow of data before an authenticated flow from a legitimate party; the receiving application may misinterpret both parts as a single stream with a single responsible peer.

Some papers a decade ago generated application-specific protocols for specific tasks, expressed in a session notation, and implementations for them [5,16,4], improving on a compiler for application-specific protocols [26]. More recently, a study of protocols and the goals they meet showed how application-level goals may be expressed in an extension of a language for protocol goals [40].

Rigorous reasoning about the behavior of TEEs is recognized need [43]. Sihna et al.'s Moat proved confidentiality properties of the code in an enclave [45]. [44] provided a much easier way to prove a much narrower property: Separate the code of an enclave into a fixed library and user code. The user code can be subjected to an automated control flow check, so it does not abuse the library. The library responsible for encrypted I/O and memory management is subjected to a one-time code verification. Thus, many enclaves can be proved to interact with the external world only through properly encrypted I/O.

Gollamudi and Chong [23] produce code for enclaves that respect information flow properties, although at the cost of a larger trusted computing base. Their compiler lays out multiple enclaves for different parts of a program, depending on security type annotations.

Barbosa et al. [3] develop cryptographic-style definitions for core functionalities within TEEs including key exchange, attested and outsourced computation. They prove that specific schemes, in standard crypto-style pseudocode, achieve these functionalities. Their fine-grained results come at the cost of mechanized support and clean construction of protocols and rules.

Much of the recent work complements ours, which provides proof goals for enclave code. If the local code meets these derived goals, our analysis shows that protocols and code will cooperate to achieve our overall application goals.

Conclusion. In this paper, we have illustrated, by means of an example and some variants, how to combine reasoning about protocols with reasoning about their context of execution. All of our reasoning is mechanized, and we provide a complete visualization of the executions that are possible for a given scenario.

We have argued that, for attestation protocols, the rules may be divided into hardware rules, trust rules, and attestation rules. This provides an objective set of requirements for the supporting mechanisms, based in hardware for attestation or in trust anchors or trust between organizations. We believe that the modular layers we found provides a repeatable way to ensure user-level protocols are crafted to their trust and attestation context.

References

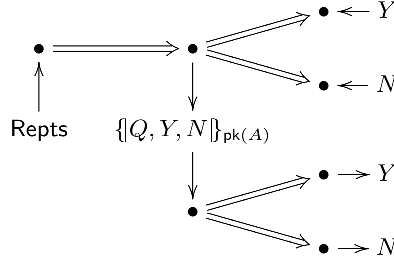
1. Martín Abadi, Michael Burrows, Butler Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–34, September 1993.
2. Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. In *Proceedings, 1994 IEEE Symposium on Research in Security and Privacy*, pages 122–136. IEEE, IEEE CS Press, 1994.
3. Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. In *IEEE EuroS&P*, pages 245–260, 2016.
4. Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *IEEE Computer Security Foundations Symposium*, 2009.
5. Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Andrew D. Gordon. Secure sessions for web services. *ACM Trans. Inf. Syst. Secur.*, 10(2):8, 2007.
6. Bruno Blanchet. An efficient protocol verifier based on Prolog rules. In *14th Computer Security Foundations Workshop*, pages 82–96. IEEE CS Press, June 2001.
7. Bruno Blanchet. *Vérification automatique de protocoles cryptographiques: modèle formel et modèle calculatoire.* Mémoire d'habilitation à diriger des recherches, Université Paris-Dauphine, November 2008.
8. Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebr. Program.*, 75(1):3–51, 2008.
9. Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, pages 537–554, 2006.
10. Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies*, 2017.
11. Ernie Brickell and Jiangtao Li. Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. In *ACM workshop on Privacy in the Electronic Society*, pages 21–30. ACM, 2007.
12. Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.
13. Rohit Chadha, Vincent Cheval, Ştefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. *ACM Trans. Comput. Log.*, 17(4):23:1–23:32, 2016.
14. Vincent Cheval. APTE: an algorithm for proving trace equivalence. In *TACAS*, pages 587–592, 2014.
15. George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. *International Journal of Information Security*, pages 1–19, 2011.
16. Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5):573–636, 2008.

17. Cas Cremers and Sjouke Mauw. *Operational semantics and verification of security protocols*. Springer, 2012.
18. Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. Completeness of the authentication tests. In J. Biskup and J. Lopez, editors, *ESORICS*, number 4734 in LNCS, pages 106–121. Springer-Verlag, September 2007.
19. Daniel Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.
20. Daniel J. Dougherty, Joshua D. Guttman, and John D. Ramsdell. Security protocol analysis in context: Computing minimal executions using SMT and CPSA. In *Integrated Formal Methods*, pages 130–150. Springer, 2018.
21. Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. *Foundations of Security Analysis and Design V*, pages 1–50, 2009.
22. Cédric Fournet, Andrew Gordon, and Sergei Maffeis. A type discipline for authorization policies. In Mooly Sagiv, editor, *European Symposium on Programming*, volume LNCS No of *LNCS*. Springer Verlag, 2005.
23. Anitha Gollamudi and Stephen Chong. Automatic enforcement of expressive security policies using enclaves. In *OOPSLA*, pages 494–513, 2016.
24. 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.
25. Joshua D. Guttman. Establishing and preserving protocol security goals. *Journal of Computer Security*, 22(2):201–267, 2014.
26. Joshua D. Guttman, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Programming cryptographic protocols. In Rocco De Nicola and Davide Sangiorgi, editors, *Trust in Global Computing*, number 3705 in LNCS, pages 116–145. Springer, 2005.
27. Joshua D. Guttman, F. Javier Thayer, Jay A. Carlson, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Trust management in strand spaces: A rely-guarantee method. In David Schmidt, editor, *Programming Languages and Systems: 13th European Symposium on Programming*, number 2986 in LNCS, pages 325–339. Springer, 2004.
28. Intel®. Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx`, 2016.
29. Intel®. Intel Software Guard Extensions (Intel SGX) data center attestation primitives: ECDSA quote library API. `https://download.01.org/intel-sgx/dcap-1.0/docs/SGX_ECDSA_QuoteGenReference_DCAP_API_Linux_1.0.pdf`, August 2018.
30. David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. `https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`, April 2016.
31. Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.
32. Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings, 2002 IEEE Symposium on Security and Privacy*, pages 114–130. May, IEEE CS Press, 2002.
33. Moses D. Liskov, Paul D. Rowe, and F. Javier Thayer. Completeness of CPSA. Technical Report MTR110479, The MITRE Corporation, March 2011. `http://www.mitre.org/publications/technical-papers/completeness-of-cpsa`.

34. Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification (CAV)*, pages 696–701, 2013.
35. Toby Murray and P. C. van Oorschot. Formal proofs, the fine print and side effects. In *IEEE SecDev*, Sept 2018.
36. 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.
37. John D. Ramsdell and Joshua D. Guttman. CPSA4: A cryptographic protocol shapes analyzer with geometric rules. The MITRE Corporation, 2018. `https://github.com/ramsdell/cpsa`.
38. John D. Ramsdell, Joshua D. Guttman, and Moses Liskov. CPSA: A cryptographic protocol shapes analyzer, 2016. `http://hackage.haskell.org/package/cpsa`.
39. E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.
40. Paul D. Rowe, Joshua D. Guttman, and Moses D. Liskov. Measuring protocol strength with security goals. *International Journal of Information Security*, February 2016. DOI 10.1007/s10207-016-0319-z, `http://web.cs.wpi.edu/~guttman/pubs/ijis_measuring-security.pdf`.
41. Salman Saghafi, Ryan Danas, and Daniel J. Dougherty. Exploring theories with a model-finding assistant. In *Conference on Automated Deduction*, volume 9195 of *Lecture Notes in Computer Science*, pages 434–449. Springer, 2015.
42. Benedikt Schmidt, Simon Meier, Cas Cremers, and David A. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. *Computer Security Foundations, (CSF)*, pages 25–27, 2012.
43. Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 38–54, 2015.
44. Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification methodology for secure isolated regions. In *PLDI*, pages 665–681, 2016.
45. Rohit Sinha, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *ACM CCS*, pages 1169–1184, 2015.
46. Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *ACM CCS*, pages 2421–2434, 2017.
47. Ting Yu, Marianne Winslett, and Kent E. Seamons. Interoperable strategies in automated trust negotiation. In *ACM CCS*, pages 146–155, 2001.

A Using the Crowbar

In this appendix, we provide—for the curiosity of referees—full details and CPSA runs for an example. They show the correctness of the recipe for application-level protocols given at the end of Section 3 in this case. They also show how



Where Repts take the forms: $[\![\,\mathsf{cert\ fm}\ f,\ \mathsf{vk}(f)\,]\!]_{\mathsf{sk}(CA)}$,
$[\![\,\mathsf{fm}\ eid_C :: ch_C :: k_C :: rest_C\,]\!]_{\mathsf{sk}(f)}$, $[\![\,\mathsf{cb}\ \ er\,]\!]_{k_C^{-1}}$.

Fig. 5. The Yes-or-No protocol

to build in additional properties such as a requirement for a fresh attestation rather than a "canned" one. Figures 6–7 are screenshots of actual CPSA output, which is quite close to the more print-ready figures in the body.

Again, input and output files have been placed on the web, and may be retrieved by referees with the help of the PC chairs.

The analysis in Section 3.3 tells us just how to use the crowbar and fulcrum. Before executing the main protocol, a role must collect the certificate $[\![\,\mathsf{cert\ fm}\ f,\ \mathsf{vk}(f)\,]\!]_{\mathsf{sk}(CA)}$, the fulcrum report $[\![\,\mathsf{fm}\ eid :: ch :: k :: rest\,]\!]_{\mathsf{sk}(f)}$, and the crowbar report $[\![\,\mathsf{cb}\ \ er\,]\!]_{k^{-1}}$.

Moreover, P needs a trust anchor CA, i.e. a $\mathsf{vk}(CA)$ such that $\mathsf{sk}(CA)$ will be used only in accordance with the protocol, meaning $\texttt{Non}(\mathsf{sk}(CA))$. Second, Rule 4 requires that the certificate for a principal f ensures that $\texttt{Non}(\mathsf{sk}(f))$, and Rule 5 requires that the code in control of $\mathsf{sk}(f)$, by using the PKI correctly, will connect only to correct attestation servers, and will use freshly generated challenge values. Finally, P must know suitable values ch such that $\texttt{CbCode}(ch)$, where the latter ensures that an enclave controlled by ch will protect its private key k^{-1} (Rule 6) and will not migrate k^{-1} (Rule 7).

A.1 The Yes-or-No protocol

As an example application level protocol, consider the *Yes-or-No* protocol, as shown in Fig. 5. In this protocol, the client P (a.k.a. the *poser*) uses the certificate, fulcrum report, and crowbar report—written jointly as Repts—to ensure that its intended peer A is compliant. It then transmits a yes/no question Q together with two nonces Y and N encrypted with $\mathsf{pk}(A)$. The job of the compliant answerer A is to release either the first nonce Y in case the answer is *yes* or else the second nonce N in case the answer is *no*. If P completes the branch receiving Y, P learns one answer, and P learns the other answer by completing the other branch.

The secrecy goal of this protocol is to ensure that even an adversary that can guess what question Q will be asked cannot determine what the answer is. The adversary cannot distinguish

Run 1 in which the answer was *yes*; v_0 was chosen as the value of the parameter Y; and v_1 was chosen as the value of the parameter N; from

Run 2 in which the answer was *no*; v_1 was chosen as the value of the parameter Y; and v_0 was chosen as the value of the parameter N.

Distinguishing these two runs would require distinguishing $\{\!|Q, v_0, v_1|\!\}_{\mathsf{pk}(A)}$ from $\{\!|Q, v_1, v_0|\!\}_{\mathsf{pk}(A)}$. With a *semantically secure* encryption, this is intractable.

Our CPSA analysis concentrates on the authentication property, which is that when P completes along either branch, the answerer must in fact have executed the corresponding branch. If the poser thinks the answer was *yes*, then the answerer really committed to *yes*; and likewise for *no*.

A.2 Rules for the client protocol

Suppose that P obtains the Repts, where in the crowbar report $[\![\, \mathsf{cb}\ er \,]\!]_{k^{-1}}$ we have $er = eid_A :: ch_A :: k_A :: rest_A$, i.e. the value er has the right structure for an enclave record.

The analysis of Section 3.3 tells us that the behavior summarized in Fig. 4 must be present, and moreover two enclaves are present, one running the crowbar itself, and the other, which the crowbar has attested, running code ch_A. Thus, for some pmk:

$$\mathtt{EnclCodeKey}(eid_C, ch_C, k_C, pmk) \quad \text{and} \quad \mathtt{EnclCodeKey}(eid_A, ch_A, k_A, pmk).$$

We only need one additional attestation rule for this protocol to be useful. This rule gives the consequences for an enclave running the code ch_A we expect for the answerer. The rule is an *attestation rule*, as it is entirely analogous to Rule 6.

Rule 10 (Answerer attestation) $\quad \forall\, eid, ch \colon \textsc{mesg}\,.$

$$\begin{aligned} &\mathtt{EnclCodeKey}(eid, ch, k, pmk) \wedge \mathtt{AnsCode}(ch) \\ &\qquad \Longrightarrow \quad \mathtt{Non}(k^{-1}). \end{aligned}$$

That is, code whose hash satisfies `AnsCode` will, if running in an enclave, generate a fresh keypair (k, k^{-1}) and successfully protect k^{-1}, using it only in accordance with the protocol.

A.3 Protocol analysis: Application level

Consider now a scenario in which a poser runs the *yes* branch to completion; in particular, it contains code hash ch_c in the fulcrum report and ch_a in the crowbar report. Moreover, we assume:

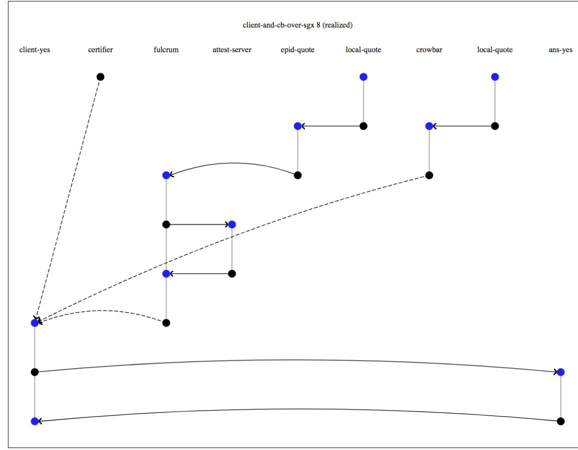


Fig. 6. CPSA output for client protocol.

Facts: $\texttt{CbCode}(ch_C)$, $\texttt{AnsCode}(ch_A)$, $\texttt{Neq}(ch_C, ch_A)$;
Keys: $\texttt{Non}(\mathsf{sk}(CA))$.

Now CPSA constructs the diagram shown in Fig. 6. The leftmost strand starts by receiving the certificate, the fulcrum report, and the crowbar report. The middle reconstructs the consequences in Fig. 4. Using $\texttt{EnclCodeKey}(eid_A, ch_A, k_A, pmk)$, which we infer, together with the assumption $\texttt{AnsCode}(ch_A)$, we apply Rule 10, inferring $\texttt{Non}(k_A^{-1})$, i.e. $\texttt{Non}(\mathsf{dk})$.

Hence, only an answerer strand can extract Y from $\{|Q, Y, N|\}_{\mathsf{pk}(A)}$; the adversary does not have the decryption key. Thus, CPSA infers the rightmost strand.

The analysis in the *client-no* case corresponds exactly.

Omitting rules. Omitting Rule 10 has the expected effect: Without it, CPSA has no ground to infer $\texttt{Non}(k_A^{-1})$. If the key k_A^{-1} is compromised, perhaps the adversary has used it to decrypt $\{|Q, Y, N|\}_k$, and the adversary can transmit Y back to the poser P. Thus, the rightmost strand in Fig. 6, the *ans-yes* strand, will not be added. The poser has no evidence of the authenticity of the answer.

A.4 Recency for the crowbar report

Fig. 6 indicates reason to think that either run of the *local-quote* role was recent at the time when the *client-yes* role occurred. The quotes may have occurred long before P poses the question Q. This is what we expect for the quote that attests to the crowbar's status, which is checked by the fulcrum. We said that this should occur once when the processor is received, to confirm the supply chain from manufacturer to purchaser.

However, possibly the crowbar should have run recently, obtaining a new attestation for the answerer enclave after the poser started his strand.

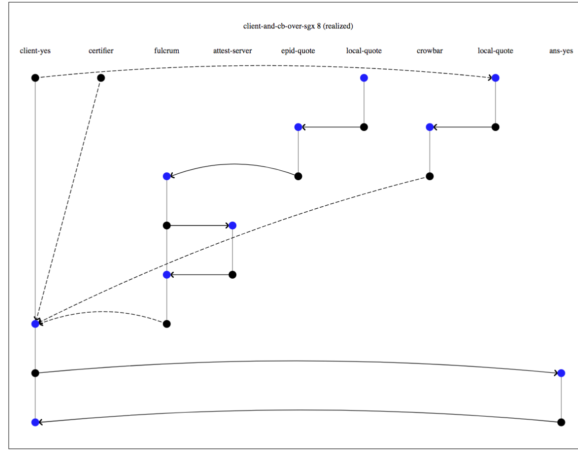


Fig. 7. CPSA output for client protocol with recency.

To obtain this functionality, no change is needed to the machinery in Sections 2 and 3. We simply make a small change to the application level protocol described in this section. We prepend a transmission to the beginning of the poser roles. It sends a fresh nonce a that the poser checks is contained as an additional field in the crowbar record attestation to the answerer. This enclave record has the form $er = eid_A :: ch_A :: k_A :: rest_A$, where the poser now checks that $rest_A = a :: rest'$ in fact contains the nonce.

In the implementation, additional functionality in the answerer enclave is needed to receive a and insert it into the enclave record so that the quote will have this form. However, our formalization is not sensitive to how this is done: any way to accomplish this is acceptable, as long as it preserves Rule 10, and the key remains non-compromised, to be used only in accordance with the protocol.

With this change to the poser protocol, CPSA produces the form shown in Fig. 7. The dashed arrow at the top records the conclusion that the poser's first transmission precedes the *local-quote* attesting to the answerer. This illustrates the flexibility of our fulcrum/crowbar machinery.