# 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. Conclusions about attestations thus depend jointly on protocols, hardware services, and program behavior.
We present a mechanized approach to modeling these properties, combining protocol analysis with axioms, that formalize hardware and software properties. Here, we model aspects of Intel's SGX mechanism. Above the underlying manufacturer-provided protocols, we build a modular user-level that uses its attestations to make trust decisions.

## 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 overall security goals? In achieving their goals, the protocols may rely on specific assumptions about the remaining components' behaviors. These assumptions define security-relevant specifications for the remaining components, focusing the design and validation processes for these 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 [33], which we have enriched with the ability to apply axioms or, as they are also called, *rules* [32]. CPSA with axioms checks if a protocol is using its context correctly. The analysis codifies what matters about this context, focusing attention—for further formal or empirical investigation—on whether the components satisfy the axioms. The axioms CPSA allows are implications, specifically universally quantified implications belonging to the *geometric fragment* of first order logic [14]. They formalize behavioral assumptions on the software and hardware context.

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 [18]. CPSA can also "read off" a strongest security goal (e.g. authentication or confidentiality) that holds for that scenario [35].

Other protocol analysis tools (e.g. Tamarin [29] and ProVerif [4]) can support variants of our method, which seems to increase its value. For instance, Tamarin's *restrictions* specify axioms, leading Tamarin to explore the set of traces that are compatible with the restrictions. This is used successfully on a substantial scale in Sapic to model protocols that manipulate state [25].

**Attesting to Trusted Execution Environments.** We illustrate our method 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 associated with other data $D$. Attestations, also called *quotes*, require support from the processor that must guarantee the TEE.

As we use TEEs, the data $D$ always includes a public key $K$. The TEE generates the key pair $K, K^{-1}$ on startup, and protects the private part $K^{-1}$, inserting $K$ 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 $C$. If the code $C$ faithfully implements a protocol $\Pi$, then $E$ uses the private key $K^{-1}$ only in accordance with that protocol $\Pi$.

TEEs—as threads with protected memory within user-level processes—are available on recent Intel processors. These so-called *enclaves* use the instruction set extension Software Guard Extensions (SGX) [22]. TEEs, as virtual machines, are available on AMD processors (Secure Encrypted Virtualization [24]). Other manufacturers may offer TEEs; academic work such as Sancus [31], for embedded systems, and Sanctum [10] also provide TEEs. Our methods are applicable well beyond SGX, which currently has weaknesses [7].

**Case study.** Our case study justifies layering substantial mechanisms, using protocol analysis and assumptions about hardware and software.

At the lower level, we represent the original mechanisms for SGX attestation, which involve complications, such as online interaction with an Intel attestation server. We identify three axioms that jointly characterize what the 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 lower layer, we illustrate how to use its attestations to draw conclusions about a user-layer protocol.

**Contributions.** We show how to combine rules and protocol analysis to design protocols targeted to hardware and software contexts. Our method provides simple descriptions of what the protocol requires from these contexts.

The rules in our case study fit three patterns that appear to be reusable for many attestation mechanisms.

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

**Trust rules** formalize the decisions and practices of an organization about creating keys and certificates, and using the certified keys.

**Attestation rules** 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 rules, or refute them [30]. Attestation rules furnish 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. [9].

CPSA is an excellent interactive tool for determining the relevant rules. We derived the ones in this paper by observing what CPSA could *not* establish. We then introduced successive rules that would provide it with information it needs, respecting the intentions of the hardware and system designers.

Our work is a descendent of *authentication logics* [26], i.e. special-purpose logics for system designers to determine trust relations. Subsequent work showed how to use standard logics (Datalog as in [27]), and how to connect them with protocols [20,16]. We add a clear axiomatic structure for the combined analysis.

A *non-contribution* of this paper is any evidence that the rules are true. Instead, we identify simple, relevant rules 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 rules. Subramanyan et al. propose one basis for reasoning about enclaves in this complementary area [40].

**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. A summary of CPSA appears in Section 2.3. Section 3 shows how an application level protocol can use SGX reliably. Sections 2.4 and 3 present the analysis at the successive layers, determining what the protocols can do subject to the rules. Overall patterns in these rules are discussed in Section 4, with related work and conclusions in Section 5.

The new CPSA is available [32], as are input and output files for this case study [21]. Our main notation is in Table 1.

**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 assertion $\mathtt{Non}(K)$ has two parts. The first is that no entity other than the intended one(s) possesses and can use the key $K$. Hardware and software must 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.

| | |
|---|---|
| $\#(m)$ | is the result of a hash function applied to $m$; |
| $\mathsf{mac}(m, K)$ | is a keyed hash or Message Authentication Code in which $K$ is the key and $m$ is the value being authenticated; |
| $mdk$ | is the MAC key on a processor, regarding $mdk$ as naming the processor. |
| $\{\lvert m \rvert\}_K$ | is 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 $\mathsf{tag}$. |
| $(K, K^{-1})$ | 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 $\mathsf{dk}(A)^{-1} = \mathsf{pk}(A)$.

**Table 1.** Notation

The term of art "non-compromised" will cover these two parts.

The 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.

**The adversary.** CPSA works within a Dolev-Yao model [12], so we always assume that the adversary controls how messages are routed among participants. The adversary can also generate values, concatenate them and separate their parts, and can encrypt and decrypt using keys it possesses or can obtain. The adversary can obtain all long term secrets we do not assume non-compromised. The adversary can guess random values unless we assume them fresh and unguessable ("uniquely originating" in CPSA's terminology).

Software and hardware can misbehave at the convenience of the adversary, except when we make explicit behavioral assumptions expressed in rules.

Thus, limitations on the adversary are under the control of the modeler.

**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* $\Pi$ is a finite set of strands called the *roles* $\rho \in \Pi$. The roles contain *parameters*, and the *instances* of $\rho$ are the strands that result from $\rho$ by plugging in values for these parameters. This set of instances—obtained from $\Pi$'s roles by plugging in values for parameters—are the *regular strands* of $\Pi$.

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

A strand may contain only an initial segment of the nodes of a role. For instance, at a particular time, a local run of the *local-quote* role (Fig. 1) may have received a message, but given as yet no response. 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 [18].

## 2 Attestation in SGX

Intel's SGX attestation mechanism involves four elements.

First, a *local quote* about a *subject enclave* $\sigma$ can be verified by a *target enclave* $\tau$ resident on the same processor. The local quote is a Message Authentication Code (MAC) prepared keyed with a secret $\#(mdk, \tau)$ hashed from $\tau$ plus a unique secret $mdk$ permanently protected within each processor (the Master Derivation Key, in SGX-speak). The content of the MAC is the *Enclave Record* (ER) for $\sigma$. The ER includes a hash of the code controlling $\sigma$ and other data. The EREPORT instruction creates a local quote.

To check a local quote, $\tau$ executes instruction EGETKEY to obtain the MAC key $\#(mdk, \tau)$, and recomputes the MAC value. Enclave $\tau$ must be resident on the same processor, because $mdk$ is used in the key $\#(mdk, \tau)$. A misbehaving $\tau$ cannot use this to forge local quotes targeted at a compliant $\tau'$, since $\tau'$, obtaining a different key $\#(mdk, \tau')$, will reject the forged MAC.

Second, to obtain attestations for entities on other devices, a *remote quote* is made by a particular enclave, the *quoting enclave* $\tau_q$. It receives a claimed ER and a local quote $q$. It checks ER and $q$ via EGETKEY. On success, it generates a digital signature on ER using the group signature scheme EPID [6].

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 generate 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 $mdk$. In

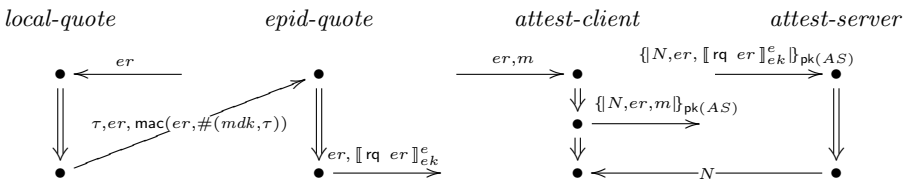



**Fig. 1.** SGX core roles

the EPID-quote role, the quoting enclave makes sure that its initial input has the form shown by executing EGETKEY on $mdk$. 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 [6]. 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. Thus, the client may possibly submit any message $m$. If the client 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

Analyzing the manufacturer's protocol uses three rules. Each one codifies what follows when a role in Fig. 1 occurs. To express them, we use predicates that say when a strand is an instance of the roles, 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 $\rho$, we write:

$\texttt{LocQt}(z, i)$ if $\rho$ is the *local-quote* role;
$\texttt{EpidQt}(z, i)$ if $\rho$ is the *epid-quote* role; and
$\texttt{AttServ}(z, i)$ if $\rho$ 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 $z$;
$\texttt{LocQtPr}(z, mdk)$ if $mdk$ is the processor secret;
$\texttt{EpidQtKey}(z, ek)$ if $ek$ is the signing EPID key of *epid-quote* instance $z$;
$\texttt{EpidQtProc}(z, mdk)$ if $z$ runs on the processor with secret $mdk$; and
$\texttt{ASQtKey}(z, ek)$ if *attestation server* run $z$ validates a quote signed with $ek$.

The rules also use uninterpreted predicate symbols. Their formal significance comes from the rules, which allow us to infer them, or infer further consequences from them. Their informal English descriptions relate them to the actual properties of the components they constrain.

**Content of the rules.** The *local-quote* role executes on a valid SGX processor only if there is an SGX-protected enclave with the given enclave record $er$. It should be 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 $er = eid :: ch :: k :: rest$.

The processor secret $mdk$ can "name" the processor. Even if no one knows $mdk$, we can still reason about whether $mdk = mdk'$, etc. A run of the *local-quote* role on a processor with non-compromised $mdk$ ensures there is an enclave with the parameters $eid, ch, k, mdk$, which we will write $\texttt{EnclCodeKey}(eid, ch, k, mdk)$:

**Rule 1** Quote guarantees enclave

$$\forall z : \text{STRD},\ eid, ch, rest : \text{MESG},\ k : \text{AKEY},\ mdk : \text{SKEY}.\ \texttt{LocQt}(z, 2) \wedge \texttt{LocQtER}(z, eid :: ch :: k :: rest) \wedge \texttt{LocQtPr}(z, mdk) \wedge \texttt{Non}(mdk) \implies \texttt{EnclCodeKey}(eid, ch, k, mdk).$$

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

When the Attestation Server completes a run, what must hold? It has checked that the purported EPID signature was genuine, and the signing key $ek$ generated interactively with the manufacturer's EPID master secret. It also vouches that the enclave mechanism can preserve the secrecy of $ek$ within the EPID quoting enclave.[1] Hence:

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

$$\forall z : \text{STRD},\ ek : \text{AKEY}.\ \texttt{AttServ}(z, 2) \wedge \texttt{ASQtKey}(z, ek) \implies \texttt{ManuMadeEpid}(ek) \wedge \texttt{Non}(ek).$$

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

The third rule applies 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 manufactured by Intel, which also generated a non-compromised processor secret $mdk$. Moreover, the processor is capable of preserving the secrecy of $mdk$ and ensuring that it is used only in accordance with the roles shown in Fig. 1. The conclusion is simply $\texttt{Non}(mdk)$, stating that $mdk$ is non-compromised, again enabling further protocol analysis.

**Rule 3** Manufacturer-made EPID on non-compromised processor

$$\forall z : \text{STRD},\ ek : \text{AKEY},\ mdk : \text{SKEY}.\ \texttt{EpidQt}(z, 2) \wedge \texttt{EpidQtKey}(z, ek) \wedge \texttt{EpidQtProc}(z, mdk) \wedge \texttt{ManuMadeEpid}(ek) \implies \texttt{Non}(mdk).$$

[1] Since an out-of-order execution attack falsifies this claim [7], the current SGX does not satisfy our axioms.

### 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 $\mathtt{Non}(\mathtt{dk}(AS))$, 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 run starts with a scenario, 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, such as $\mathtt{Non}(\mathtt{dk}(AS))$ and $\mathtt{Unique}(N)$. 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 [18]. To find them, CPSA explores increasingly 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 [12], starting with initial values compatible with assumptions such as $\mathtt{Non}(\mathtt{dk}(AS))$ and $\mathtt{Unique}(N)$.
2. Suppose $\eta$ instantiates the variables of a rule $R$ to make the hypothesis of $R$ true. Then $\eta$ yields a true instantiation of the conclusion of $R$.

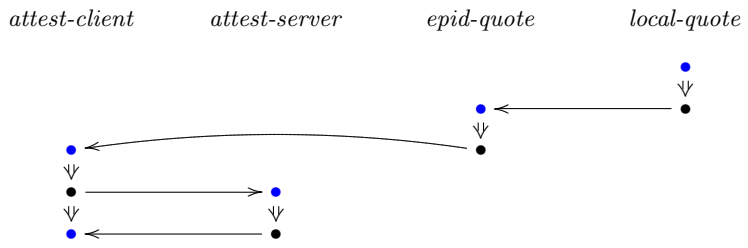
CPSA uses *authentication tests* [18] to find a small set of enrichments to explain a message reception that does not yet satisfy Clause 1. CPSA considers how to add new regular strands and new hypotheses about compromised keys. Alternative possible explanations cause branching in the search.

When $R$ and $\eta$ are a counterexample to Clause 2, CPSA adds information to make the conclusion hold. When the conclusion is an equation $s = t$, CPSA equates the values $\eta(s)$ and $\eta(t)$. When the conclusion is an atomic formula $P(t_1, \ldots, t_k)$, it adds its instance $P(\eta(t_1), \ldots, \eta(t_k))$ to the scenario.

The approach accommodates additional forms of conclusion, containing *conjunctions*, *existentially quantifiers* $\exists x \,.\, \phi$, and *disjunctions* (logical *or*s, although not in this paper). These syntactic forms are preserved by all homomorphisms [19], and the rules are *geometric sequents* [14]. This yields scenarios that cover all possible executions that homomorphically enrich the initial scenario [18,36,13].

CPSA is implemented in Haskell, and the core program takes input in s-expression format, and gives its output as s-expressions. This is then converted by several supplementary tools to other forms, especially `xhtml` to be displayed in a browser.

**CPSA's input and output.** Given a protocol and rules, CPSA's input is a scenario consisting of some strands of regular participants, together with assumptions such as $\mathtt{Non}(\mathtt{dk}(AS))$ and $\mathtt{Unique}(N)$ or other facts (closed atomic formulas). The starting scenario and similar structures are called *skeletons*.




**Facts:** $\texttt{ManuMadeEpid}(ek)$, $\texttt{EnclCodeKey}(eid, ch, k, mdk)$
***Non* keys:** $\texttt{Non}(mdk)$, $\texttt{Non}(ek)$, $\texttt{Non}(\texttt{dk}(AS))$

**Fig. 2.** Consequences of an attestation client success

CPSA returns skeletons representing all minimal, essentially different executions enriching the initial skeleton. This is the empty set when the initial skeleton cannot occur; e.g. it hypothesizes some security disclosure that cannot occur. Very often, this set is small, containing only one or a few possibilities.

CPSA presents its results by diagrams like those in Figs. 2, 4, etc., in `xhtml`. 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 $ek$ and $mdk$ 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 $\texttt{ManuMadeEpid}(ek)$ and $\texttt{Non}(ek)$.

**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 $mdk$ parameter was previously unknown. By Rule 3, we infer $\texttt{Non}(mdk)$.

**3.** How was the local quote $\mathtt{mac}(er, \#(mdk, \tau))$ 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 $mdk$. This fact is expressed by $\mathtt{EnclCodeKey}(eid, ch, k, mdk)$.

The analysis is now complete.

**Omitting rules.** Omitting Rule 1 does not change the diagram, but the fact $\mathtt{EnclCodeKey}(eid, ch, k, mdk)$ 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 $mdk$.

Omitting Rule 3 omits this fact, as well as the (rightmost) *local-quote* strand. The key $mdk$ is no longer known to be $\mathtt{Non}$. Finally, omitting Rule 2 means that only the *attest-server* strand is available. Each rule has a 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 with fresh $N$ and non-compromised $\mathtt{dk}(AS)$: 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 $k$.

These conclusions depend on the rules: SGX hardware should ensure that a local quote on a processor with non-compromised $mdk$ 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 $mdk$ non-compromised (Rule 3).

Making the three rules hold requires challenging—not yet fully achieved—engineering [7]. However, the rules summarize the requirements succinctly, transparently, and usefully for mechanized analysis.

## 3 An application protocol using quotes

Section 2 shows how to use SGX attestation. To learn $\mathtt{EnclCodeKey}(eid, ch, k, mdk)$, we attempt a Attestation Client local run with $er = eid :: ch :: k :: rest$. If we succeed with fresh nonce $N$ and non-compromised peer key $\mathtt{Non}(\mathtt{dk}(AS))$, then $\mathtt{EnclCodeKey}(eid, ch, k, mdk)$ holds for some $mdk$ with $\mathtt{Non}(mdk)$.

Suppose software analysis of the code with hash $ch$ shows this code will:

1. generate a fresh key pair $k, k^{-1}$;
2. place the public value $k$ into the its enclave record, while protecting the private value $k^{-1}$; and
3. use $k^{-1}$ only for the cryptographic operations required by certain roles of $\Pi$, and only under the control structure required by those roles.

Hence $\mathtt{Non}(k^{-1})$; so we can use $k$ to contact enclave $eid$ for those roles of $\Pi$. In this section, we work out an example protocol, using a rule that summarizes

the software analysis (1)–(3), plus a rule that expresses the client's policy of checking an attestation before engaging in the application level protocol $\Pi$.

As an example application level protocol, consider the *Yes-or-No* protocol, as shown in Fig. 3. In this protocol, the client $P$ (a.k.a. the *poser*) sends 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. Before asking its question, the poser obtains a valid enclave record of the form $eid :: ch :: \mathsf{pk}(as) :: rest$, abbreviated $er$ in Fig. 3.

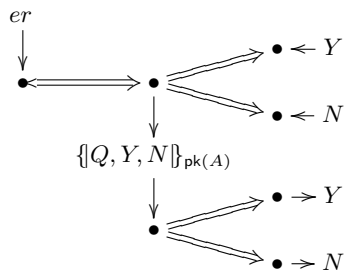



**Fig. 3.** The Yes-or-No protocol

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*. There is an additional secrecy goal, 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.

**An Attestation Rule for the Peer.** Suppose the predicate $\texttt{AnsCode}(ch)$ holds true only of bitstrings that result by compiling and hashing source code that we have analyzed. Suppose, moreover, that our analysis indicates this code satisfies properties (1)–(3), specifically when "certain roles of $\Pi$" means the two answerer roles of the Yes-or-No protocol. Unless other entities discover its private key $k^{-1}$, $k^{-1}$ will be non-compromised, i.e. known only to a principal that will use it only in accordance with the protocol. Moreover, the processor prevents other entities from discovering $k^{-1}$ from an enclave, as it encrypts evicted memory. This justifies a rule:

**Rule 4** (Answerer attestation) $\quad \forall\, eid, ch : \text{MESG}\,.$

$$\texttt{EnclCodeKey}(eid, ch, k, mdk) \wedge \texttt{AnsCode}(ch) \implies \texttt{Non}(k^{-1}).$$

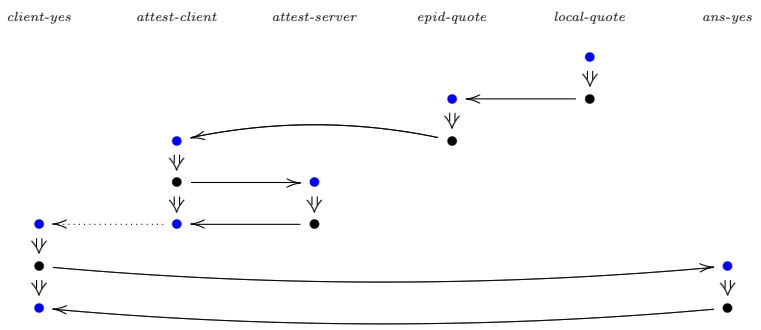



**Fig. 4.** CPSA output for client protocol, *yes* branch.

**Client rule: Obtain quote first.** Suppose parties executing the poser roles in the Yes-or-No protocol obtains an attestation for its peer before starting the run. Perhaps its code ensures a control flow in which the code implementing the poser roles cannot be reached until after an attestation is complete. We can express this via a pair of rules, using the predicates $\texttt{PoseY}(s, i)$ and $\texttt{PoseN}(s, i)$ to mean that strand $s$ is an instance of the poser affirmative or negative role, up to height (step) $i$. The parameter predicates $\texttt{Eid}(s, eid)$, $\texttt{Ch}(s, ch)$, $\texttt{Ans}(s, a)$, and $\texttt{Rest}(s, rest)$ to indicate strand $s$ is given an enclave record with components (respectively) $eid$ for the EID; $ch$ for the controlling code hash; $a$ for the answerer peer; and $rest$ for the remainder. The following rule asserts that the start of an affirmative poser strand must be preceded by a successful attestation client run with suitable $er$:

**Rule 5** (Client gets quote) $\forall\, s\colon \text{STRD},\ eid, ch, rest\colon \text{MESG},\ a\colon \text{NAME}\,.$

$$\begin{aligned} & \texttt{PoseY}(s, 1) \quad \wedge \texttt{Eid}(s, eid) \wedge \texttt{Ch}(s, ch) \wedge \texttt{Ans}(s, a) \wedge \texttt{Rest}(s, rest) \\ \Longrightarrow \quad & \exists\, z\colon \text{STRD}, n\colon \text{TEXT},\ as\colon \text{NAME}\,.\, \texttt{AttCl}(z, 3) \wedge \texttt{AttClN}(z, n) \wedge \\ & \qquad \texttt{AttClAs}(z, as) \wedge \texttt{AttClEr}(z, eid :: ch :: \mathsf{pk}(as) :: rest) \wedge \\ & \qquad (z, 2) \prec (s, 0) \wedge \texttt{Non}(\mathsf{pk}(as)) \wedge \texttt{Unique}(n) \end{aligned}$$

A symmetric rule—about a negative poser strand $\texttt{PoseN}(s_{no}, 1)$ of height at least 1—is not needed, since any negative poser strand agrees with a positive strand up to height 1 (and 2). Thus, Rule 5 applies to $s_{no}$ also.

**Protocol analysis: Application level.** Suppose now a poser runs the *yes* branch to completion, with challenge nonce $Y$, code hash $ch$, and peer public key $\mathsf{pk}(as)$. Moreover, assume:

$$\textbf{Fact:}\ \ \texttt{AnsCode}(ch) \qquad\qquad \textbf{Keys:}\ \ \texttt{Unique}(Y).$$

Now CPSA constructs the diagram shown in Fig. 4. The leftmost strand starts by receiving the enclave record. Rule 5 ensures that there is a *attest-client* run that

precedes it, indicated by the dotted arrow. The middle reconstructs the consequences in Fig. 2. Using $\texttt{EnclCodeKey}(eid, ch, \mathsf{pk}(as), mdk)$, which Fig. 2 implies, and the assumption $\texttt{AnsCode}(ch)$, CPSA applies Rule 4, inferring $\texttt{Non}(\mathsf{dk}(as))$.

Hence, only an answerer strand can extract $Y$ from $\{|Q, Y, N|\}_{\mathsf{pk}(as)}$; 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 4 has the expected effect: Without it, CPSA has no grounds to infer $\texttt{Non}(\mathsf{dk}(as))$. If the key $\texttt{Non}(\mathsf{dk}(as))$ 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. 4, the *ans-yes* strand, will not be added. The poser has no evidence of the authenticity of the answer.

Omitting Rule 5 means that no other strands need to be added. Without an attestation, nothing is known about $\texttt{Non}(\mathsf{dk}(as))$.

## 4 Types of rules

We first categorize Rules 1–3 from Section 2 and Rules 4–5 from Section 4. We divide them 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 $mdk$, 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 $ek$ unless the processor can protect its secret $mdk$.

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 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 $ek$ 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 5 expresses the client's policy of always checking the remote attestation for $er$ before asking a question $Q$. Again, there is no attestation here, since there is no evidence that particular code is in control of the client.

**Attestation rules.** Rule 4 is an attestation rule. It applies only when its premise $\texttt{EnclCodeKey}(eid, ch, k, mdk)$ is known to hold, i.e,. other evidence has already established the existence of an enclave $eid$ with code (hashing to) $ch$.

A rational process—analyzing the behaviors of the code with known hash $ch$—governs proposed enclave rules. Does it randomly generate its keypair $k, k^{-1}$ and install $k$ 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 respect the control flow of the specific roles

in which this key is expected to engage? In attestation rules, we always know what code is in control of an enclave.

**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 few seconds on a standard laptop. CPSA is competitive with other symbolic protocol protocol analysis tools (e.g. Scyther [11] and Tamarin [29]) for a variety of examples [28], and seems much faster than some (e.g. Maude-NPA [15]).

The rules we have developed are modeled on the intent of the SGX mechanisms. However, we expect that other attestation mechanisms, based on different hardware primitives, will lead to alternative sets of rules. The use of those rules for protocol analysis, however, will be very similar in cases that we can envisage.

**The danger of unsound rules.** What prevents an analyst from writing wishful-thinking rules rather than accurate descriptions of the intended system?

Nothing, in fact. The rules require scrutiny, to determine whether they are reasonable specifications of the system ingredients. We recommend that rules be reused whenever possible, so that when reusing components in a new design one should reuse the rules that have already passed scrutiny as specifying it. We offered our taxonomy of rules into hardware, trust, and attestation rules as a way to focus attention on the main jobs that rules fulfill in this application area. This is meant to encourage rule specifiers to write rules that have a clear, simple explanation. If they are wrong, there is more likely to be a well-known property of the system, or a way to test the system, to expose the error.

**Uses of the rules.** The rules provide *specifications* of the relevant components. Hardware rules make clear what we need from SGX. Trust rules provide guidelines for organizations' public attestation server and client code. 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, or whether to adopt existing components. *Testing* gets improved focus from these succinct, intuitive rules. The testers should especially attempt to identify whether these rules could be wrong.

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 a jackpot for the red team, namely the failure of the mechanism.

**Alternate protocols and rules.** We have considered a variety of different protocols, including Intel's newer, EPID-free ECDSA protocol [23], our own mechanism which uses standard signatures on top of EPID, and simplifications of the protocols that omit tags such as rq in Fig. 1. Additional rules are required in these cases, but they reuse the same ideas we have presented already.

A small change to the application-level protocol also assures the client that the attestation occurred recently.

## 5 Related work and Conclusion

**Security protocol analysis** is a very well-developed field, with numerous sophisticated tools for trace properties (e.g. [4,15,29,33]), and some for determining indistinguishability properties also (e.g. [5,8]). In many cases, our work is compatible with other approaches rather than in competition with it. For instance, tamarin [29] has a notion of *restriction* used to restrict the traces of interest. 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, which is specific to CPSA, is useful in development. It provides an overview of all minimal, essentially different possibilities.

**Connecting security protocols to context** has been less studied than one would expect. Protocol failures such as the renegotiation attacks on TLS [34] arise because the protocol does not provide enough information to its context when the authenticated identity of the peer changed.

Some papers a decade ago generated application-specific protocols for specific tasks, expressed in a session notation, and implementations for them [3,2], improving on a compiler for application-specific protocols [20]. 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 [35].

**Rigorous reasoning about the behavior** of TEEs is a recognized need [37]. Sihna et al.'s Moat proved confidentiality properties of the code in an enclave [39]. [38] provided a much easier way to prove a much narrower property: Separate the code of an enclave into a fixed library and user code. And automated control flow check on the user code ensures it does not abuse the library. The library can be 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. A more general model may now be found in [40]; it uses a clean state machine transition model to formalize core integrity, confidentiality, and attestation properties that SGX and other TEE models such as Sanctum [10].

Gollamudi and Chong [17] produce code for enclaves that respect information flow properties, although at the cost of a larger trusted computing base.

Barbosa et al. [1] 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. In particular, they do not identify anything similar to the contrast of hardware, trust, and attestation, as their goals are more aligned with cryptographic mechanisms.

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.** We have illustrated, by means of example, how to combine reasoning about protocols with reasoning about their context of execution. All of our reasoning is mechanized, with a visualization of the executions for each scenario.

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. Modular layers provide a repeatable way to ensure user-level protocols are crafted to their trust and attestation context.

## References


1. 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.
2. 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.
3. Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Andrew D. Gordon. Secure sessions for web services. *ACM Trans. Inf. Syst. Secur.*, 10(2):8, 2007.
4. Bruno Blanchet. An efficient protocol verifier based on Prolog rules. In *IEEE CSFW*, pages 82–96. IEEE CS Press, June 2001.
5. Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology - CRYPTO 2006*, pages 537–554, 2006.
6. 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.
7. 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.
8. 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.
9. G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of remote attestation. *IJIS*, 2011.
10. Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.
11. Cas Cremers and Sjouke Mauw. *Operational semantics and verification of security protocols*. Springer, 2012.
12. Daniel Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.
13. 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.
14. Roy Dyckhoff and Sara Negri. Geometrisation of first-order logic. *Bulletin of Symbolic Logic*, 21(2):123–163, 2015.
15. 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.
16. Cédric Fournet, Andrew Gordon, and Sergei Maffeis. A type discipline for authorization policies. In *European Symposium on Programming*, LNCS, 2005.

17. Anitha Gollamudi and Stephen Chong. Automatic enforcement of expressive security policies using enclaves. In *OOPSLA*, pages 494–513, 2016.
18. 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.
19. Joshua D. Guttman. Establishing and preserving protocol security goals. *Journal of Computer Security*, 22(2):201–267, 2014.
20. 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*, LNCS, pages 116–145. Springer, 2005.
21. Joshua D. Guttman and John D. Ramsdell. CPSA inputs for understanding attestation, April 2019. `https://web.cs.wpi.edu/~guttman/pubs/understanding_attestation_example/`.
22. Intel. Intel® Software Guard Extensions (Intel® SGX). `https://software.intel.com/en-us/sgx`, 2016.
23. Intel. Intel® Software Guard Extensions (Intel® SGX) data center attestation primitives: ECDSA quote library API. `https://download.01.org/intel-sgx/dcap-1.0.1/docs/Intel_SGX_ECDSA_QuoteGenReference_DCAP_API_Linux_1.0.1.pdf`, November 2018.
24. 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.
25. Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *Journal of Computer Security*, 24(5):583–616, 2016.
26. 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.
27. 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.
28. Moses D. Liskov, Joshua D. Guttman, John D. Ramsdell, Paul D. Rowe, and F. Javier Thayer. Enrich-by-need protocol analysis for Diffie-Hellman. In *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows*, pages 135–155, 2019.
29. 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.
30. Toby Murray and P. C. van Oorschot. Formal proofs, the fine print and side effects. In *IEEE SecDev*, Sept 2018.
31. 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.
32. 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`.
33. John D. Ramsdell, Joshua D. Guttman, and Moses Liskov. CPSA: A cryptographic protocol shapes analyzer, 2016. `http://hackage.haskell.org/package/cpsa`.
34. E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), 2010.

35. 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`.
36. 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.
37. 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 S&P*, pages 38–54, 2015.
38. 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*, 2016.
39. Rohit Sinha, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *ACM CCS*, 2015.
40. Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *ACM CCS*, 2017.