A Verification Framework for Stateful Security Protocols

Li Li[1], Naipeng Dong[2], Jun Pang[3], Jun Sun[1], Guangdong Bai[2], Yang Liu[4], and Jin Song Dong[2,5]

[1]Singapore University of Technology and Design, [2]National University of Singapore, [3]University of Luxembourg, [4]Nanyang Technological University, [5]Griffith University

Abstract. A long-standing research problem is how to efficiently verify security protocols with tamper-resistant global states, especially when the global states evolve unboundedly. We propose a protocol specification framework, which facilitates explicit modeling of states and state transformations. On the basis of that, we develop an algorithm for verifying security properties of protocols with unbounded state-evolving, by tracking state transformation and checking the validity of the state-evolving traces. We prove the correctness of the verification algorithm, implement both of the specification framework and the algorithm, and evaluate our implementation using a number of stateful security protocols. The experimental results show that our approach is both feasible and practically efficient. Particularly, we have found a security flaw on the digital envelope protocol, which cannot be detected with existing security protocol verifiers.

1 Introduction

Automatic formal verification is shown to be extremely useful in analyzing security protocols. Many security protocol verifiers have been developed, for instance, ProVerif [1], AVISPA [2] and Maude-NPA [3]. However, such verifiers fail in analyzing security protocols with shared objects such as databases, registers and memory locations [4]. Real-world examples include protocols involving security devices like IBM's 4758 CCA secure coprocessor platform and trusted platform module (TPM) [5] and protocols involving databases for websites and key servers [6].

As these shared objects must be maintained externally w.r.t. sessions, the objects are abstracted as global states; and protocols with these shared objects are refereed to as stateful protocols. The global states have three properties: (1) *mutable*: the value of a state can be updated, (2) *unbounded evolving*: the value updating of a state can be unbounded, and (3) *tamper-resistant*: the value of a state can only be updated by legitimate users. For instance, the following example is a simple stateful protocol, where the security device is a shared object, i.e., a global state.

Example 1. Consider a security device *SD* (a variation of [6]), with a tamper-resistant memory initialized to a constant '$init$'. *SD* supports three public operations: (1) reading: the current value stored in the memory can be read; (2) updating: the memory with current value m, can be updated to $\mathsf{h}(m, x)$, where h is a hash function and x is an arbitrary value; (3) decrypting: when receiving a ciphertext of the form $\mathsf{enc_a}(\langle m_f, s_l, s_r\rangle, pub)$, i.e., a sequence of three values $\langle m_f, s_l, s_r\rangle$ asymmetrically encrypted by *SD*'s public key pub, *SD* decrypts it. According to *SD*'s current memory value m, it continues as follows: if $m = \mathsf{h}(m_f, \mathit{left})$, *SD* sends out s_l; if $m = \mathsf{h}(m_f, \mathit{right})$, *SD* sends out s_r, where left and right are two publicly known constants. Suppose *Bob*, a legitimate user, generates two secrets s_l and s_r, reads the memory of *SD* as m_f and sends a ciphertext $\mathsf{enc_a}(\langle m_f, s_l, s_r\rangle, pub)$ to *SD*. The *SD* ensures that a malicious *Bob* or any other attackers can never know both 's_l' and 's_r' at the same time, since *SD* cannot be configured as both '$\mathsf{h}(m_f, \mathit{left})$' and '$\mathsf{h}(m_f, \mathit{right})$' in one execution.

Verification of stateful protocols has been noticed as important and necessary but challenging [5] even for a simple protocol as Example 1. In particular, ProVerif – one of the popular and widely used verifier (e.g., used in [7–10]), reports false attacks for some stateful protocols such as Example 1. Recently, an extension StatVerif is proposed, which is specialized in verifying stateful protocols [6]. However, StatVerif can produce false attacks when the state-value mutates (e.g., when the security device in Example 1 reboots) and cannot terminate when the state-value mutates unboundedly (e.g., when the protocol in Example 1 keeps running).

We improve the Horn clause based verification (used in ProVerif and StatVerif) for analyzing stateful protocols with unbounded global state evolving (i.e., unbounded evolving steps with potentially unbounded values of a state). Horn clause reasoning is inherently monotonic – once an event (the basic element in Horn clause) is true, it cannot be set to false anymore, and thus does not work well for state-value mutation in ProVerif and StatVerif [4]. Therefore, we propose to distinguish global states from events. In particular, we explicitly model global states and their evolving transformations in specification. More importantly, on each step of reasoning in verification, we record the state-evolving constraints; and when a target event is derived, we instantiate a state-evolving trace satisfying the constraints in the derivation, i.e., the global states can evolve following the trace such that the derivation could happen. In such a way, we reduce the false attacks caused by global states' unbounded evolving.

For example, we model the security device as a global state $SD(\_)$, which consists of two parts: the name of the object (SD) used to distinguish different objects and its pre-defined fields ('_') used to distinguish attributes of the same object (the field '_' indicates the memory of the security device). Each field is filled with a concrete value of the attribute at any time, e.g., the memory field can be filled with 'init', $\mathsf{h}(m_f, \mathit{left})$, or $\mathsf{h}(m_f, \mathit{right})$. Hence, $SD(\mathit{init})$, $SD(\mathsf{h}(m_f, \mathit{left}))$, and $SD(\mathsf{h}(m_f, \mathit{right}))$ are the possible *instantiations* of the global state $SD(\_)$. A particular instantiation of a global state may be visited multiple times in one trace of the global state's evolving. To distinguish each appearance of an instantiation, we additionally add a distinct index a_i to the instantiation, and require all indexes in a trace to have chronological orders. We name an instantiation of a global state and its index a *snapshot*. The snapshots of a global state must form an evolving trace starting from an initial instantiation, based on the index's chronological order. We allow variables to appear in the snapshot to represent a set of snapshots, and name the snapshot with variables a snapshot *pattern*.

In verification, we explicitly validate the evolving traces of the snapshots. Suppose the adversary obtains message m_1 and m_2 at the following snapshot pattern respectively

$$\big(SD(\mathsf{h}(\mathsf{h}(x_1, x_2), \mathit{left})), a_1\big), \big(SD(\mathsf{h}(\mathsf{h}(\mathsf{h}(\mathit{init}, \mathit{right}), x_1'), \mathit{left})), a_2\big),$$

Type	Expression			
Message(m)	$a[], b[], A[], B[], \perp$	(name)	$[n], [k], [N], [K]$	(nonce)
	x, y, z, X, Y, Z	(variable)	$\mathsf{f}(m_1, m_2, ..., m_n)$	(function)
Guard(g)	$m_1 \not\mapsto m_2$	$(\nexists \sigma, m_1 \cdot \sigma = m_2)$	$m_1 \neq m_2$	(inequivalence)
Event(e)	$know(m)$	(knowledge)	$new([n], l[])$	(generation)
	$init(m_1, \cdots, m_n)$	(initialization)	$accept(m_1, \cdots, m_n)$	(acceptance)
	$leak(m)$	(leakage)		
State(s)	$name(id_1, \cdots, id_s, m_1, \cdots, m_n)$	(state)		

Table 1: Syntax Hierarchy

where variables x_1, x_2 and x'_1 can be arbitrary values, a_1 and a_2 are indexes of the two snapshots (any ordering is possible). In order to conduct the attack that the adversary obtains both m_1 and m_2, the adversary tries to find an instantiation of the variables x_1, x_2 and x'_1 such that a valid trace exists for the security device to evolve from its initial snapshot $(SD(init), a_0)$ to the above snapshots. We can see that the following evolving trace exists, when $x_1 = \mathsf{h}(init, right)$, and $x_2 = x'_1$ can be an arbitrary value,

$$SD(init) \to SD(\mathsf{h}(init, right)) \to SD(\mathsf{h}(\mathsf{h}(init, right), x_2)) \\ \to SD(\mathsf{h}(\mathsf{h}(\mathsf{h}(init, right), x_2), left)).$$

That is, the adversary tries to guide the protocol to perform the above global state transformation, and then obtains both m_1 and m_2 at the last snapshot. However, if an additional snapshot $\big(SD(\mathsf{h}(init, left)), a_3\big)$ exists e.g., for the adversary to obtain m_3, then no valid evolving trace exists for the adversary to obtain m_1, m_2 and m_3, since the device memory cannot be set to $SD(\mathsf{h}(init, left))$ and $SD(\mathsf{h}(init, right))$ (contained in the snapshot with index a_2) no matter in which order in a single trace. Hence, the attack is infeasible for obtaining all three pieces of information.

We introduce the formal modeling of global states and their transformations in the subsequent section, then propose our verification algorithm in Section 3, and finally present our experimental results in Section 4 and discuss related works in Section 5.

2 Protocol Specification

To verify whether a protocol satisfies a security property, an analyzer needs to formally specify the protocol (without states in Section 2.1) and the property (Section 2.3). The key part is how the global states and state transformations are formalized (Section 2.2).

2.1 Preliminary – Specification Syntax without States

As in most verifiers, messages – the basic elements in protocols, are modeled by names, nonces, variables and functions (first row in Table 1). Names model constants; nonces are freshly generated random numbers; variables represent memory locations for holding messages, and functions can be applied to a sequence of messages. All messages are assumed to be well-typed and variables can be instantiated only once.

The relations between messages are as follows. A message containing variables can be instantiated by a *substitution*, e.g., $\sigma = \{x_1 \mapsto m_1, \cdots, x_n \mapsto m_n\}$ instantiates the variables $x_1, \cdots, x_n$ with the messages $m_1, \cdots, m_n$ respectively. Given two messages m_1 and m_2, when there exists a substitution σ such that $m_1 \cdot \sigma = m_2$, we say that m_1 *is unified to* m_2, denoted as $m_1 \rightsquigarrow_\sigma m_2$. When m_1 should not be unified to m_2, we write $m_1 \not\rightsquigarrow m_2$. For instance, when a message m should not be a tuple, we write $m \not\rightsquigarrow \langle m_1, m_2 \rangle$. Given two messages m_1 and m_2, if there exists a substitution σ such that $m_1 \cdot \sigma = m_2 \cdot \sigma$, we say m_1 *and* m_2 *are unifiable* and σ is a unifier of m_1 and m_2, denoted as $m_1 =_\sigma m_2$. If m_1 and m_2 are unifiable, the most general unifier of m_1 and m_2 is a unifier σ such that for any unifier σ' of m_1 and m_2 there exists a substitution σ'' such that $\sigma' = \sigma \cdot \sigma''$. When m_1 and m_2 should not be unifiable (a.k.a., inequivalence), we write $m_1 \neq m_2$. For instance, if the current branch condition is that the protocol responder r is not *Bob*, we write $r \neq Bob$. $m_1 \not\rightsquigarrow m_2$ and $m_1 \neq m_2$ form the guarding conditions (second row in Table 1) i.e., whether an rule (defined later) can be applied.

Based on the above definitions, a protocol is modeled as a set of logical rules, similar as in ProVerif [1] and Tamarin [11]. The basic elements of a rule are events. An event is applying a predicate to a message sequence. The following two events are used in the protocol specification:

- event $know(m)$ means that the adversary knows the message m; and the
- event $new([n], l[])$ models that a nonce $[n]$ (the concrete value of the nonce) is freshly generated at the location $l[]$ (symbolic value used to distinguish the nonce from other nonces in a specification) by a legitimate protocol participant. Note that nonce $[n_1] \ldots [n_k]$ with the same location $l[]$ are k concrete generation of the same nonce specification in k different sessions.

The intuition is that a protocol and its involved cryptographic primitives can be treated as oracles accessible to the adversary. The adversary having the required messages obtains the corresponding outputs. Once receiving an input, the oracle generates nonces, processes messages and outputs messages according to its specification. Each oracle is modeled as a rule $[\ G\ ]\ H \ -\![\quad]\!\!\rightarrow e$, where G is a set of guard conditions, H is a set of premise events, and e is a conclusion event, meaning that if the guard conditions in G and the premise events in H are satisfied, then the conclusion event in e is satisfiable.

Cryptographic primitives. The premises of a cryptographic primitive are a set of $know$ events specifying the input parameters, and the conclusion is one $know$ event representing the generated result, e.g., the asymmetric encryption and decryption used in Example 1 is modeled as follows, where m, pub and sk are variables.

$$know(m), know(pub) \ -\![\quad]\!\!\rightarrow know(\mathsf{enc_a}(m, pub)) \tag{1}$$

$$know(\mathsf{enc_a}(m, \mathsf{pk}(sk))), know(sk) \ -\![\quad]\!\!\rightarrow know(m) \tag{2}$$

Protocol. A pair of the message input and the subsequent output of a participant are specified as an oracle as well. The difference is that we need to additionally consider the nonce generation and potential guard conditions. Whenever a nonce at position $l[]$ is generated in a protocol, we model the nonce generation by adding a $new([d], l[])$ event to H of the oracle. Whenever $m_1 \neq m_2$ or $m_1 \not\rightsquigarrow m_2$ conditions are required (which rarely happen in protocol specification) in the current execution branch, we add the conditions into G. For example, *bob*'s behavior in Example 1 can be modeled as

$$\begin{aligned} &new([bob_l], l_{s_l}[]), new([bob_r], l_{s_r}[]), know(m_f) \ -\![\quad]\!\!\rightarrow \\ &\qquad know(\mathsf{enc_a}(\langle m_f, [bob_l], [bob_r] \rangle, \mathsf{pk}(sksd[])))\end{aligned} \tag{3}$$

2.2 Protocol Specification with States

As addressed in the introduction, we explicitly model the global states of a protocol as well as their transformations. There are two ways that the states are involved. First, we use snapshots to represent at which state a rule can be applied. Second, we use a rule to model how the state transforms.

For the first case, we introduce a set of snapshots S into the rule to denote the involved states, use M to record at which snapshot each event happens (each element in M is of the form $e_i :: a_j$ with $e_i \in H$ and a_j being the index of a snapshot in S), and use O to denote the constraints on chronically orders between snapshots (each element in O is of the form $a_i \mathcal{R}\, a_j$ with a_i and a_j being indexes of snapshots in S). We define three types of ordering relations between two snapshots a_i and a_j in $\mathcal{R}$: (1) $a_i \leq a_j$ means that a_i appears earlier than a_j; (2) $a_i \lessdot a_j$ means that the shared object is modified once between a_i and a_j; (3) $a_i \sim a_j$ means that the shared object remains unchanged between a_i and a_j. A rule now is of the form $[\ G\ ]\ H\ :\ M\ \dashv\!\![\ S\ :\ O\ ]\!\!\mapsto e$ where e is an event. We name such rules as *state consistent rules*. For example, depending on the configuration, the *SD* replies s_l or s_r in Example 1, and the behavior of replying s_l is modeled as follows:

$$\begin{aligned} &\mathit{know}(\mathsf{enc_a}(\langle m_f, s_l, s_r\rangle, \mathsf{pk}(\mathit{sksd}[])))^{\textcircled{1}} : \{\textcircled{1} :: a_1\} \dashv\!\![\ \big(\mathit{SD}(\mathit{init}[]), a_0\big), \\ &\qquad\qquad \big(\mathit{SD}(\mathsf{h}(m_f, \mathit{left}[])), a_1\big) : \{a_0 \leq a_1\}\ ]\!\!\mapsto \mathit{know}(s_l) \end{aligned} \tag{4}$$

where ① is a reference to the corresponding premise event in H, so that we do not need to repeat the entire event in M, in order to save space and have a clearer presentation. The rule describes that if (1) the *SD* reads in a ciphertext $\mathsf{enc_a}(\langle m_f, s_l, s_r\rangle, \mathsf{pk}(\mathit{sksd}[]))$ at snapshot a_1, which is denoted by an event $\mathit{know}(\mathit{enc_a}(\langle m_f, s_l, s_r\rangle, \mathit{pk}(\mathit{sksd}[])))$, and the mapping between the event and a set of snapshots $\textcircled{1} :: \{a_1\}$ where ① refers to the know event; and (2) snapshot a_1 is reachable, i.e., there should be a valid trace from the initial state $\mathit{SD}(\mathit{init}[])$ to the current state $\mathit{SD}(\mathsf{h}(m_f, \mathit{left}[]))$, which is denoted by the two snapshots $\big(\mathit{SD}(\mathit{init}[]), a_0\big)$, $\big(\mathit{SD}(h(m_f, \mathit{left}[])), a_1\big)$ and their ordering constraints $\{a_0 \leq a_1\}$, meaning that a_0 needs to appear earlier than a_1 in a trace; then (3) the *SD* returns s_l, since the current configuration is $\mathsf{h}(m_f, \mathit{left}[])$. Another type of state consistent rule is that the adversary may be able to obtain information from the states, e.g., the reading operation in Example 1 can be modeled as

$$\dashv\!\![\ \big(\mathit{SD}(\mathit{init}[]), a_0\big), \big(\mathit{SD}(m), a_2\big) : \{a_0 \leq a_2\}\ ]\!\!\mapsto \mathit{know}(m) \tag{5}$$

meaning that if a_2 is reachable (denoted by $\big(\mathit{SD}(\mathit{init}[]), a_0\big), \big(\mathit{SD}(m), a_2\big) : \{a_0 \leq a_2\}$ with a_0 being the initial state), then the adversary can read the current value in the memory, modeled as $\mathit{know}(m)$.

For the second case, we introduce the *state transferring rules* of the form $[\ G\ ]\ H :\ M\ \dashv\!\![\ S\ :\ O\ ]\!\!\mapsto T$ where T is a set of state transformations (a sequence of two snapshots). For example, the *SD* can be updated in Example 1, which is modeled as

$$\begin{aligned} &\mathit{know}(x)^{\textcircled{2}} : \{\textcircled{2} :: a_3\} \dashv\!\![\ \big(\mathit{SD}(\mathit{init}[]), a_0\big), \big(\mathit{SD}(m), a_3\big) : \{a_0 \leq a_3\}\ ]\!\!\mapsto \\ &\qquad\qquad \langle \big(\mathit{SD}(m), a_3\big), \big(\mathit{SD}(\mathsf{h}(m, x)), a_4\big)\rangle \end{aligned} \tag{6}$$

meaning that the adversary who has x at state $\mathit{SD}(m)$ can update the *SD* to be $\mathsf{h}(m, x)$, where $\langle \big(\mathit{SD}(m), a_3\big), \big(\mathit{SD}(\mathsf{h}(m, x)), a_4\big)\rangle$ models the transformation of *SD* from snapshot a_3 to snapshot a_4.

2.3 Security Properties

We focus on two types of security properties: authentication and secrecy. To formalize authentication properties, we add the following two events: When the protocol initiator starts a protocol run, we add a corresponding $init$ event (defined in Table 1) into H; when the protocol responder accepts a protocol run, we add a corresponding $accept$ event (defined in Table 1) into C. Then authentication is modeled as correspondence between the *init* and *accept* events (as in most verifiers such as ProVerif and StatVerif).

Definition 1 (Authentication). *In a security protocol, an authentication property holds, i.e., correspondence between an* accept *event and an* init *event with agreed arguments holds, if and only if for every occurrence of event $accept(m_1, \cdots, m_n)$, the corresponding $init(m_1, \cdots, m_n)$ event must be engaged before, and all the required snapshots form a valid evolving trace, denoted as $accept(m_1, \cdots, m_n) \Leftarrow init(m_1, \cdots, m_n)$.*

The secrecy property specifies that the adversary cannot obtain certain secret messages. It is defined by introducing a rule with the $leak$ event (defined in Table 1) as the conclusion. If secrecy is preserved in a protocol, the $leak$ event should not be reachable.

Definition 2 (Secrecy). *In a protocol, secrecy holds for a message m if and only if $leak(m)$ is not reachable after adding $new_1, \cdots, new_n, know(m) \dashv [\quad] \mapsto leak(m)$, where $new_1, \cdots, new_n$ are the nonce generation events for all nonces in m.*

Intuitively, if the adversary knows the message $know(m)$, the message m is leaked; and the new events are used to accurately specify the nonces in m.

As commonly assumed, we consider an active network attacker who can intercept all communications, compute new messages, generate new nonces and send the messages he obtained. For computation, he can use all the publicly available functions, e.g., encryption, decryption and concatenation. He can also designate honest participants to initiate new protocol runs and to take part in the protocol whenever he needs to.

3 Verification Algorithm

Given a set of rules $\mathcal{B}_{init}$ specifying a protocol (including stateless rules, state consistent rules and state transferring rules) and a property as described in Section 2, the verification aims to find the derivations of the target event specified in the property ($accept$ event for authentication and $leak$ for secrecy) using the rules in $\mathcal{B}_{init}$, and then check whether a derivation contradicts the specified property.

To derive a target event using a set of rules, directly reasoning on the rules would not terminate, e.g., repeatedly applying Rule (1) leads to increasingly complex terms [1, 6]. To improve efficiency and help termination, we follow the approach in [1, 6] – providing an algorithm to guide the reasoning. Hence, similar to [1], we construct a rule base $\mathcal{B}$, in the first phase, by combining pairs of rules in $\mathcal{B}_{init}$, which may infer new rules. Then we perform query searching in $\mathcal{B}$ to find valid attacks in the second phase. The key idea of our rule-base construction is as follow: If a rule's premise events are trivially satisfiable (events in $\mathcal{N}$), we can use its conclusion to fulfill other rules' complex premises (events not in $\mathcal{N}$). This is called *rule composition*. By applying rule composition repeatedly on existing rules until saturation, we can then safely remove the rules with complex premise events, because whenever the rule with complex premises is used in the reasoning, it can be replaced with an alternative rule (often generated by composition) with all premise events in $\mathcal{N}$. In addition, when a new rule is inferred by rule composition, *rule implication* operation is applied to check whether this rule is necessary to be added to $\mathcal{B}$. If the new rule is implied by existing rules then it is not necessary to add it. These two operations are shown to be efficient in avoiding complex terms and accelerating the verification process in ProVerif.

We generally follow the above procedure as proposed in ProVerif, but we need to add snapshot trace validation in rule composition and rule implication. Intuitively, rule composition applies one rule after another. Thus, regarding states, we ensure that (1) the snapshot ordering constraints in both rules are still preserved and (2) the ordering between the two rules are added to the ordering constraints in the resulting new rule. For rule implication, regarding states, we need to define that the ordering constraints of snapshots in a rule is less than the constraints in another one, i.e., whenever the second rule is applicable, the first rule is also applicable.

In addition, we try to concretize the snapshot traces in a rule if possible, to narrow the possible traces satisfying the ordering constraints in the rule, since it is sufficient as long as one trace exists to reach the conclusion event. To do so, we introduce two additional operations: *state unification* and *state transformation*. The intuition is as follows: Any two snapshots appear in a rule may have three kinds of relations: (1) they are from different objects; (2) they are of the same object, and the object is not modified between two snapshots; (3) they are of the same object, and the object is modified between the two snapshots. In the first case, we do not need to search for a valid trace between the two snapshots. In the second case, we try to unify them to the same value, i.e., *state unification*. In the third case, we try to find the transformations between them, i.e., *state transformation*. Note that these two operations only need to be applied to rules (1) with its premise events in $\mathcal{N}$, since those with premise event not in $\mathcal{N}$ will be eventually removed; and (2) with their conclusion events be $leak$ event or $accept$ event, since they are the query goals.

3.1 Preliminary Definitions

We first define the set $\mathcal{N}$ as the following three types of events, similar to ProVerif: (1) initializing a new protocol (an $init$ event), (2) generating a fresh nonce (a new event), (3) knowing an arbitrary value (a $know(x)$ event where x is a variable).

Recall that a rule may contain a set of ordering constraints O specified using relation $\mathcal{R}$ defined in Section 2, we define O as *closed* if the following properties hold.

$$\begin{array}{ll}
a_i \lessdot a_j, a_j \sim a_k \in O \Rightarrow a_i \lessdot a_k \in O & a_i \lessdot a_j \in O \Rightarrow a_i \leq a_j \in O \\
a_i \lessdot a_j, a_k \sim a_j \in O \Rightarrow a_i \lessdot a_k \in O & a_i \sim a_j \in O \Rightarrow a_i \leq a_j \in O \\
a_i \sim a_j, a_j \lessdot a_k \in O \Rightarrow a_i \lessdot a_k \in O & a_i \sim a_j, a_j \sim a_k \in O \Rightarrow a_i \sim a_k \in O \\
a_j \sim a_i, a_j \lessdot a_k \in O \Rightarrow a_i \lessdot a_k \in O & a_i \leq a_j, a_j \leq a_k \in O \Rightarrow a_i \leq a_k \in O
\end{array}$$

In verification, we first ensure the O in every rule is closed using the above definition. Given two sets of ordering constraints O and O', we use $O \uplus O'$ to denote their closed union. When all snapshots of the same object are connected by $\lessdot$ and $\sim$ in an acyclic trace (i.e., no uncertain relation $\leq$), we conclude that a valid evolving trace is found.

Let $R = [\,G\,]\,H : M \dashv [\,S : O\,] \mapsto V$ and $R' = [\,G'\,]H' : M' \dashv [\,S' : O'\,] \mapsto V'$ be two rules. (1) *'R having less restricted mappings than R''* means that if some premise events in R are required to be satisfied at a snapshot (s, a_j), the same premise events need to be satisfied at an earlier snapshot (s', a_i) in R' ($a_i \le a_j$). This indicates that R' has more restrictions on the satisfaction of the premises than R. This requirement can be formally captured by the joint operator '$*$'.

$$M * O = \{\langle e_i, a_j\rangle \mid e_i :: a_k \in M \wedge a_j \le a_k \in O\}$$

For every event e_i, $M * O$ captures all the snapshots later than the snapshot at which the event should be satisfied. The larger the set $M * O$ is, the earlier the event e_i needs to be satisfied. Hence, $(M * O) \subseteq (M' * O')$ captures that R has less restricted mappings. (2) *'R having more organized ordering than R''* means that for every two snapshots (s_1, a_i) and (s_2, a_j) appearing in both R and R', the ordering of the two snapshots in R is more concrete (less uncertain) than in R'. Since, $a_i \lessdot a_j$ or $a_i \sim a_j$ is more concrete than $a_i \le a_j$, given an ordering O, we measure its uncertainty (less concrete) with

$$\delta(O) = \{a_i \le a_j \mid a_i \le a_j \in O\} - \{a_t \le a_k \mid a_t \lessdot a_k \in O \vee a_t \sim a_k \in O\}.$$

O is more organized than O' if and only if $\delta(O) \subseteq \delta(O')$. $\delta(O)$ captures the uncertain ordering relations between every two snapshots in the snapshot set O. The larger the set $\delta(O')$ is, the more uncertain the ordering O' is, and hence the less organized R' is.

3.2 Rule Operations

Similar to ProVerif, when the premise of a rule contains an event not in $\mathcal{N}$, we try to fulfill/unify the event with a conclusion of other state consistent rules whose premises are in $\mathcal{N}$ by rule composition.

Definition 3 (Rule Composition). *Let $R = [\,G\,]\,H : M \dashv [\,S : O\,] \mapsto e$ be a state consistent rule and $R' = [\,G'\,]\,H' : M' \dashv [\,S' : O'\,] \mapsto V$ be either a state consistent rule or a state transferring rule. If there exists $e_0 \in H'$ such that $e =_\sigma e_0$, then R with R' can be composed on the event e_0, and the newly composed rule is defined as*

$$R \circ_{e_0} R' = \big([\,G \cup G'\,](H \cup (H' - \{e_0\})) : M \cup M' \cup M_0 \\ \dashv [\,(S \cup S') : O \uplus O' \uplus O_0\,] \mapsto V\big) \cdot \sigma,$$

$M_0 = \{e_i :: a_k \mid e_i \in H, e_0 :: a_k \in M'\}$, $O_0 = \{a_i \le a_j \mid (s, a_i) \in S, e_0 :: a_j \in M'\}$.

In the resulting rule $R \circ_{e_0} R'$, the guard condition $G \cup G'$, premise events $H \cup (H' - \{e_0\})$ and conclusion event V are straightforward, following the same idea as in ProVerif. Regarding states, $S \cup S'$, $M \cup M'$ and $O \uplus O'$ capture that the snapshots, event-snapshot mapping and ordering constraints in both rules need to be satisfied in the resulting rule. For event-snapshot mapping, we additionally require that any event $e_i \in H$ needs to map to the snapshots of e_0 (i.e. a_k), such that R can be applied at state a_k. Otherwise even if e and e_0 are unifiable, after applying R, R' cannot be applicable, due to that the state of e_0 is not satisfied. This requirement is captured by M_0. For the snapshot ordering, we additionally require that any snapshot in S should appear before the snapshot of e_0, capturing that R is applied before R' in order to obtain e (or e_0), and thus the snapshots of R should appear before the snapshot for e_0, as modeled in O_0.

Given two rules R and R', if R (1) has the same conclusion as R' but requires less guard conditions and less premises (the same as in ProVerif), (2) has less snapshots, less restricted mappings and more organized ordering (additional requirements regarding states), we say that R implies R', denoted as $R \Rightarrow R'$.

Definition 4 (Rule Implication). *Let $R = [\ G\ ]\ H\ :\ M\ \dashv\!\![\ S\ :\ O\ ]\!\!\mapsto\ V$ and $R' = [\ G'\ ]H'\ :\ M'\ \dashv\!\![\ S'\ :\ O'\ ]\!\!\mapsto\ V'$ be two rules. We define R implies R' denoted as $R \Rightarrow R'$ if and only if $\exists\sigma$,*

$$\begin{aligned}&(1)\big((V\cdot\sigma = V')\ \wedge\ (G\cdot\sigma\subseteq G')\ \wedge\ (H\cdot\sigma\subseteq H')\big)\ \wedge\\ &(2)\big((S\cdot\sigma\subseteq S')\ \wedge\ ((M*O)\cdot\sigma\subseteq(M'*O'))\ \wedge\ (\delta(O)\cdot\sigma\subseteq\delta(O'))\big).\end{aligned}$$

By now, we updated the rule composition and rule implication with additional requirements on states. Hereafter we introduce operations to concertize a snapshot trace.

Given two snapshots $(s_1, a_i), (s_2, a_j)$ of the same object in a rule, if s_1 and s_2 are unifiable ($s_1 =_\sigma s_2$), the simplest trace between s_1 and s_2 is to unify them as one snapshot, capturing the situation where the object is not modified between the two snapshot (formally $a_i \sim a_j$ or $a_j \sim a_i$).

Definition 5 (State Unification). *Let $R = [\ G\ ]\ H\ :\ M\ \dashv\!\![\ S\ :\ O\ ]\!\!\mapsto\ e$ be a state consistent rule. Assume there exist two distinct snapshots $(s_1, a_i), (s_2, a_j) \in S$ such that $s_1 =_\sigma s_2$, then we can unify the two snapshots in rule R; and the state unification of s_1 to s_2 on R is defined as*

$$R[a_i \sim a_j] = \big([\ G\ ]\ H : M \dashv\!\![\ S : O \uplus \{a_i \sim a_j\}\ ]\!\!\mapsto e\big)\cdot\sigma.$$

Note that if $s_1 =_\sigma s_2$, both $R[a_i \sim a_j]$ and $R[a_i \sim a_j]$ will be generated.

Given a state consistent rule $R = [\ G\ ]\ H\ :\ M\ \dashv\!\![\ S\ :\ O\ ]\!\!\mapsto\ e$, if a snapshot $(s, a_i) \in S$ does not have an immediate previous snapshot defined in O, i.e., $\nexists(s', a_j) \in S : a_j \lessdot a_i \in O \vee a_j \sim a_i \in O$, we try to apply a state transferring rule to find an immediate previous snapshot. Given a rule R, we use $\eta(S, O)$ to denote the snapshots in S whose previous snapshots have not been found, i.e.,

$$\eta(S, O) = S - \{(s, a_i) \mid a_j \lessdot a_i \in O \vee a_j \sim a_i \in O\}.$$

Definition 6 (State Transformation). *Let $R = [\ G\ ]\ H\ :\ M\ \dashv\!\![\ S\ :\ O\ ]\!\!\mapsto\ T$ be a state transferring rule and $R' = [\ G'\ ]\ H'\ :\ M'\ \dashv\!\![\ S'\ :\ O'\ ]\!\!\mapsto\ e$ be a state consistent rule. Assume there is an injective function $f : T \to \eta(S', O')$, such that $\forall t = \langle(s, a_i), (s', a_j)\rangle \in T, s' =_\sigma s''$ if $f(t) = (s'', a_k)$. The state transformation of applying R to R' on f is*

$$R \bowtie_f R' = \big([\ G \cup G'\ ]\ (H \cup H') : M \cup M' \dashv\!\![\ (S \cup S') : O \uplus O' \uplus O_0 \uplus O''\ ]\!\!\mapsto e\big)\cdot\sigma,$$

where $O_0 = \{a_i \lessdot a_k \mid \langle(s, a_i), (s', a_j)\rangle \in T, f(t) = (s'', a_k)\}$, and $O'' = \{a_t \leq a_i\ | a_t \leq a_k \in O', t = \langle(s, a_i), (s', a_j)\rangle \in T, f(t) = (s'', a_k)\}$.

O_0 captures that for a state transformation $\langle(s, a_i), (s', a_j)\rangle$ in T and a function $f(t) = (s'', a_k) \in S'$, a_i did exact one transformation to a_k, because the snapshot (s', a_j) will not appear in the new rule, as it is unified with (s'', a_k). O'' enforces the snapshots (e.g., a_t) that appear earlier than a_k in O' to be also earlier than a_i in the new rule. The intuition is that there is an immediate concrete transformation from a_i to a_k ($a_i \lessdot a_k$), but the relation between a_t and a_k is rather uncertain; in this case, we try to align the three snapshots as $a_t \leq a_i \lessdot a_k$. Note it is sufficient to find one trace among a_t, a_i and a_k. Applying the above operations leads to new rules, some of which may not be valid.

Definition 7 (Rule Validation). *A rule $R = [\ G\ ]\ H : M \dashv\!\![\ S : O\ ]\!\!\mapsto V$ is valid if and only if (1) $V \notin H$; (2) O is closed and $\forall\ a_i \leq a_j \in O : a_j \leq a_i \notin O$; (3) $\forall\ know(x),\ know(y) \in H$: $x \not\equiv y$, and $\forall\ init(x), init(y) \in H$: $x \not\equiv y$, and $\forall\ new([n], l[]), new([n'], l'[]) \in H$: $n \not\equiv n' \vee l \not\equiv l'$; (4) $\forall\ e_i :: a_j \in M : e_i \in H \wedge \exists (s, a_j) \in S$, and $\forall\ a_i \mathcal{R} a_j \in O : \exists (s, a_i) \in S \wedge \exists (s', a_j) \in S$.*
The rule validation procedure of R is denoted as

$$R \Downarrow = \big(\mathsf{merge}(H) : \mathsf{clear}(M) \dashv\!\![\ S : \mathsf{clear}(O)\ ]\!\!\mapsto V\big) \cdot \sigma$$

where function merge *removes the duplicated premises, function* clear *removes references of non-existing events and snapshots, σ is the most general unifier such that any two redundant events can be merged or unified.*

We use $x \not\equiv y$ to denote that x is not syntactically equal to y. The definition says that a rule $[\ G\ ]\ H : M \dashv\!\![\ S : O\ ]\!\!\mapsto V$ (V is an event e or a set of state transformations T) is valid if and only if it satisfies: (1) If V is an event, V should not be in H; (2) O should be closed and contains no contradictory constraints; (3) there is no redundant events (two events modeling the same thing) in H (redundant events should be unified and merged); and (4) all mappings in M and all orderings in O do not involve non-existing events or snapshots (non-existing events or snapshots should be removed).

Heuristics. If provided with a snapshot pattern, we try to instantiate the snapshots in a rule with the pattern to accelerate the process of finding a concrete snapshot trace. Consider a state consistent rule $R = [\ G\ ]\ H : M \dashv\!\![\ S : O\ ]\!\!\mapsto e$, where $H \subseteq \mathcal{N}$, $e = know(x)$ and x is a variable. This implies that x does not appear in H; otherwise, the rule is not valid. Hence, x must be originated from S, for example the reading operation supported by the security device in Example 1. Since $know(x) \in \mathcal{N}$, we cannot compose R with other rules. To guide the verification, we try to apply the pattern to the states, so that R can be composed with other rules.

Definition 8 (State Instantiation). *Let $R = [\ G\ ]\ H : M \dashv\!\![\ S : O\ ]\!\!\mapsto e$ be a state consistent rule. Given a snapshot $(s, a_i) \in S$ and its pattern p such that $s =_\sigma p$, we define the state instantiation of the snapshot s with its pattern p as follows*

$$R[s \mapsto p] = \big([\ G\ ]\ H : M \dashv\!\![\ S : O\ ]\!\!\mapsto e\big) \cdot \sigma.$$

3.3 Rule Base Construction

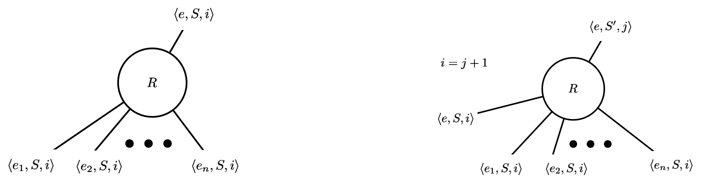
Using the above rule operations, we develop an algorithm to construct the rule base (Algorithm 1). The algorithm guides the verification by selecting proper rules to perform rule operations. Given an initial set of rules $\mathcal{B}_{init}$ as input, the algorithm returns the rule base $\mathcal{B}$ as output. In the algorithm, we first add the rules in $\mathcal{B}_{init}$ to the set $rules$ (line $8-11$). During this procedure, redundant rules are removed (line $1-6$). Then we apply rule operations on the rules in $rules$ and obtain a saturated rule set $\mathcal{B}_v$ (line $13-35$). The algorithm defines which operation is applied to which types of rules. Finally, we select those rules in $\mathcal{B}_v$ with premises in $\mathcal{N}$ and conclusion event being $accept$ or $leak$ to form $\mathcal{B}$. Now we prove the correctness of the algorithm.

Theorem 1. *Any $accept$ or $leak$ event e that is derivable from the initial rules $\mathcal{B}_{init}$ if and only if it is derivable from the knowledge base $\mathcal{B}$ constructed in Algorithm 1.*

Algorithm 1: Rule Base Construction

```
Input  : $\mathcal{B}_{init}$ - initial rules
Output: $\mathcal{B}$ - knowledge base
Procedure add (R, rules)
    for $R_b \in rules$ do
        if $R_b \Rightarrow R$ then return $rules$;
        if $R \Rightarrow R_b$ then $rules = rules - \{R_b\}$;
    end
    return $\{R\} \cup rules$;
Algorithm
    $rules = \emptyset$;
    for $R \in \mathcal{B}_{init}$ do
        $rules = add(R, rules)$;
    end
    repeat
        Case 1. Rule Composition
        Select a state consistent rule $R = H \dashv\!\![\, S : O \,]\!\!\mapsto e$
        and a general rule $R' = H' \dashv\!\![\, S' : O' \,]\!\!\mapsto V$ from $rules$ such that
        1. $H \subseteq \mathcal{N}$; 2. $\exists e_0 \in H' : e_0 \notin \mathcal{N}$;
        $rules = add((R \circ_{e_0} R') \Downarrow, rules)$;
        Case 2. State Unification
        Select a state consistent rule $R = H \dashv\!\![\, S : O \,]\!\!\mapsto e$ from $rules$ such that
        1. $H \subseteq \mathcal{N}$ and $e$ is an accept event or a leak event;
        2. $\exists s, s' \in S$, $s$ and $s'$ can be unified;
        $rules = add(R[s \sim s'] \Downarrow, rules)$;
        Case 3. State Transformation
        Select a state transferring rule $R = H \dashv\!\![\, S : O \,]\!\!\mapsto T$
        and a state consistent rule $R' = H' \dashv\!\![\, S' : O' \,]\!\!\mapsto e$ from $rules$ such that
        1. $H \cup H' \subseteq \mathcal{N}$ and $e$ is an accept event or a leak event;
        2. $\exists f, \forall t \in T, f(t) = (s, a_j), \nexists a_i \lessdot a_j \in S'$;
        $rules = add((R \bowtie_f R') \Downarrow, rules)$;
        Case 4. State Instantiation
        Select a state consistent rule $R = H \dashv\!\![\, S : O \,]\!\!\mapsto e$ from $rules$ such that
        1. $H \subseteq \mathcal{N}$, $e \in \mathcal{N}$; 2. $\exists s \in S$, $s$ has pattern $p$;
        $rules = add(R[s \mapsto p] \Downarrow, rules)$;
    until fix-point is reached;
    $\mathcal{B}_v = rules$;
    return $\mathcal{B} = \{R = H \dashv\!\![\, S : O \,]\!\!\mapsto e \in rules \mid \forall p \in H, p \in$
      $\mathcal{N} \wedge e$ is an accept event or a leak event$\}$;
```

The basic idea is as follows: Whenever there is an attack using the rules in $\mathcal{B}_{init}$, there is an attack using the rules in $\mathcal{B}_v$, since there is no rule missing. Then we only need to show that the selected rules (rules in $\mathcal{B}_v$) would not miss an attack. To do so, we first introduce the representation of an attack – the derivation tree for an $leak$ or $accept$ event from a set of rules as follows:


$\langle e, S, i\rangle$
R
$\langle e_1, S, i\rangle$
$\langle e_2, S, i\rangle$
$\langle e_n, S, i\rangle$
$\langle e, S', j\rangle$
$i = j + 1$
$\langle e, S, i\rangle$
R
$\langle e_1, S, i\rangle$
$\langle e_2, S, i\rangle$
$\langle e_n, S, i\rangle$

(a) State Consistent Rule (b) State Transferring Rule

Fig. 1: Rule in derivation tree

Definition 9 (Derivation Tree). *A closed rule is a rule with its conclusion initiated by its premises and states. Let $\mathcal{B}_t$ be a set of closed rules and e_t be an event, e_t is derivable from $\mathcal{B}_t$ if and only if there exists a finite derivation tree satisfying the following.*

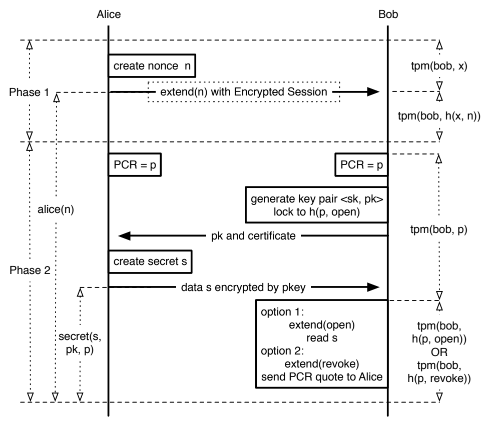
1. *Every edge in the tree is labeled by an event e, a set of snapshots $S = \{(s_1, a_1), \ldots, (s_l, a_l)\}$ and an index i, and $\forall (s_i, a_i), (s_j, a_j) \in S$: $a_i \not\sim a_j$.*
2. *Every node is labeled by a rule in $\mathcal{B}_t$.*
3. *If a node is labeled by a state consistent rule R as in Figure 1a, then we have $R \Rightarrow H : M \dashv\!\![\ S_0 \cup S : O\ ]\!\!\mapsto e$ where $H = \{e_1, \cdots, e_n\}$, M is defined as $\forall e \in H : e :: \{a_1, \ldots, a_l\}$, $O = \{a_0 \leq a_i | (s_0, a_0) \in S_0, (s_i, a_i) \in S\}$ with S_0 being the set of initial snapshot of each object; and the indexes labeled on the outgoing edge and incoming edges (Figure 1a) are the same.*
4. *If a node is labeled by a state transferring rule R as in Figure 1b, there exists a set of state transformation T such that $R \Rightarrow H : M \dashv\!\![\ S_0 \cup S : O\ ]\!\!\mapsto T$ where $H = \{e_1, \cdots, e_n\}$, M is defined as $\forall e \in H : e :: \{a_1, \ldots, a_l\}$, $O = \{a_0 \leq a_i | (s_0, a_0) \in S_0, (s_i, a_i) \in S\}$ with S_0 being the initial snapshots; let $S_{pre} = \{(s_i, a_i) | \langle (s_i, a_i), (s_j, a_j)\rangle \in T\}$ and $S_{post} = \{(s_j, a_j) | \langle (s_i, a_i), (s_j, a_j)\rangle \in T\}$, we have $S_{pre} \subseteq S$; and in Figure 1b, $S' = S - S_{pre} + S_{post}$, e can be any event that is satisfied at S, the indexes labeled on the incoming edges equal to the index labeled on the outgoing edge plus 1.*
5. *Outgoing edge of the root is labeled by the event e_t and the index 1.*
6. *Incoming edges of the leaves are only labeled by events in $\mathcal{N}$ with the same index.*
7. *The edges with the same index have the same state.*

Then we prove that whenever there is a derivation tree for an $accept$ or a $leak$ event using rules in $\mathcal{B}_v$, there is a derivation for the event using the rule base $\mathcal{B}$ created using Algorithm 1, and vice versa. The key part in the proof is the following Lemma which demonstrates how to replace two directly connected nodes in the derivation tree with one node labeled by a composite rule with the same state and index. Detailed proofs of the theorem and lemma are available online [12].

Lemma 1. *If $R_o \circ_{e_0} R'_o$ is valid, $R_t \Rightarrow R_o$ and $R'_t \Rightarrow R'_o$, then either there exists e' such that $R_t \circ_{e'} R'_t$ is valid and $R_t \circ_{e'} R'_t \Rightarrow R_o \circ_{e_0} R'_o$, or $R'_t \Rightarrow R_o \circ_{e_0} R'_o$.*

3.4 Query Searching

The query of authentication property and secrecy property is to find a rule that disproves the properties. A rule disproves non-injective authentication if and only if its conclusion



Alice
Bob
create nonce n
tpm(bob, x)
Phase 1
extend(n) with Encrypted Session
tpm(bob, h(x, n))
PCR = p
PCR = p
generate key pair <sk, pk> lock to h(p, open)
alice(n)
pk and certificate
tpm(bob, p)
create secret s
Phase 2
data s encrypted by pkey
option 1: extend(open) read s option 2: extend(revoke) send PCR quote to Alice
secret(s, pk, p)
tpm(bob, h(p, open)) OR tpm(bob, h(p, revoke))

Fig. 2: The DEP protocol

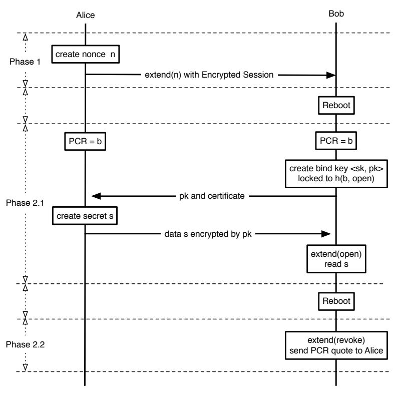


Fig. 3: An attack on DEP

event is an *accept* event, while it does not require the corresponding *init* event in its premises. A rule disproves secrecy when the $leak$ event is reachable.

Definition 10. ***Authentication Counterexample.*** *A rule $R = [\, G\,]\; H : M \dashv\!\![\; S : O \;]\!\!\mapsto e$ disproves authentication property $Q_n := accept \Leftarrow init$ denoted as $Q_n \nvDash R$ if and only if $G \neq false$, e and $accept$ are unifiable with the most general unifier σ such that $\forall e' \in H, e' \in \mathcal{N}$ and $\forall \sigma' : (init \cdot \sigma \cdot \sigma' \notin H \cdot \sigma)$.*

Definition 11. ***Secrecy Contradiction.*** *A rule $R = [\, G\,]\; H : M \dashv\!\![\; S : O \;]\!\!\mapsto e$ disproves secrecy property $Q_s := leak(m)$ denoted as $Q_s \nvDash R$ if and only if $\forall e' \in H : e' \in \mathcal{N}$, $G \neq false$, $\exists \sigma, leak(x) \cdot \sigma = e$.*

If we cannot find any counterexample during the verification, when our algorithm terminates, the protocol satisfies the property. For a detailed proof, see [12].

Theorem 2. *Let $\mathcal{B}$ be the rule base generated in Algorithm 1. When Q is a secrecy query or an authentication query, there exists R derivable from $\mathcal{B}_{init}$ such that $Q \nvDash R$ if and only if there exists $R' \in \mathcal{B}$ such that $Q \nvDash R'$.*

4 Case Studies

We have implemented the proposed approach in a tool named SSPA (Stateful Security Protocol Analyzer). Using SSPA, we have successfully verified Example 1, three versions of the digital envelope protocols [13, 7] and the Bitlocker protocol [14] to show its applicability to stateful protocols. To show that SSPA also works for protocols without global states, we have verified two versions of the Needham-Schroeder public key protocol [15, 16]. The tool detected a security flaw in the digital envelope protocol (DEP) when the trusted platform module (TPM) reset is enabled. The tool, all protocol models and their evaluation results are available online at [17]. In the remaining part, we provide more details on the DEP protocol and the detected security flaw.

DEP consists of two phases as shown in Figure 2. In the first phase, *Alice* generates a secret nonce $[n]$ and uses it to extend a given PCR in *Bob*'s TPM with an encrypted session (detailed TPM explanation can be found at [12]). Since the nonce $[n]$ is secret, *Bob* cannot re-enter the current state of the TPM if he makes any changes to the given PCR. In the second phase, *Alice* and *Bob* read the value of the given PCR as p and *Bob* creates a binding key pair $\langle [sk], \mathsf{pk}([sk]) \rangle$ locked to the PCR value $\mathsf{h}(p, \mathit{open}[])$ and sends the public binding key together with the key certificate to *Alice*, where $\mathit{open}[]$ is an agreed constant in the protocol. This means that the generated binding key can be used only if the value $\mathit{open}[]$ is first extended to the PCR of value p. After checking the correctness of the certificate, *Alice* encrypts the data $[s]$ with the public key $\mathsf{pk}([sk])$ and sends it back to *Bob*. Later, *Bob* can either open the digital envelope by extending the PCR with $\mathit{open}[]$ or revoke his right to open the envelope by extending another pre-agreed constant $\mathit{revoke}[]$. If *Bob* revokes his right, the quote of PCR value $\mathsf{h}(p, \mathit{revoke}[])$ can be used to prove *Bob*'s revoking action.

Using our approach and the implemented tool, we have found a cold-boot attack for this DEP when the TPM rebooting is allowed (see Figure 3). When the TPM rebooting is allowed, *Bob* can reboot his TPM immediately after the first phase. *Bob* can reset the PCR value, e.g., to b. As a consequence, the secret nonce $[n]$ extended to the PCR is lost. When *Alice* reads the PCR value in the beginning of the second phase, she actually reads a PCR value b that is unrelated to her previous extending action. Later this new PCR value b is used in generating key; and the key is used to encrypt *Alice*'s secret $[s]$. On receiving *Alice*'s cipher-text, *Bob* can open it and read $[s]$ by extending the PCR value by $\mathit{open}[]$. Since *Bob* is allowed to reboot, he reboots the TPM, resets the PCR value to b, and extends the PCR value by $\mathit{revoke}[]$. Now *Bob* can get a PCR quote proving that he did not open the ciphertext, despite the fact that he has opened it.

The previously DEP verification in [7] fails to detect the above attack, because, in order to use the automatic verification tool ProVerif, which can only handle limited number of TPM steps, the authors made modification to the original DEP protocol – *Bob* always performs the TPM reboot before the first phase; and *Alice* is assumed to have the PCR value $\mathsf{h}(p, n)$ without actually reading it, in the beginning of the second phase. As a result, in their model, TPM rebooting can never happen before the second phase. Hence, the modification prevents them from detecting the attack. Since we allow state modeling and unbounded state-evolving, we can remove the assumption made in [7], and thus are able to detect the flaw.

5 Related Works

Formal analysis of security protocols has been an active research area since 1980's. The analysis is with respect to the Dolev-Yao attacker [18], who controls the network by blocking, inserting, eavesdropping messages in the network. Verifying security of protocols with bounded sessions has shown to be decidable [19]; however, the verification of unbounded sessions is, in general, undecidable [20]. Verifiers that do not bound the sessions rely on abstractions that may result in false attacks, e.g., ProVerif [1], and/or allow non-termination, e.g., Maude-NPA [3]. In this work, we focus on protocols with unbounded sessions and in general follow the ProVerif style when no global states are involved. However, our work reduces false attacks when global states are involved.

For verifiers that can handle global states, StatVerif [6] is mostly relevant to our work. As mentioned earlier, StatVerif does not terminate for unbounded involving of global states (see Example 1). In addition, StatVerif still has false attacks due to the monotonicity of Horn clauses, for example, when the security device in [6] (similar protocol to Example 1 which allows the memory to be reset to a value instead of being extended to a value) is first set to either $left$ or $right$ and then set to $reboot$ (the StatVerif code for this scenario and Example 1 can be found at [17]). Our method does not have this problem for the above example protocols. Kremer et al. extended StatVerif with the ability to model unbounded number of global states [4], while our work enables to model and verify unbounded evolving of global states. Hence, our work is orthogonal to the work of Kremer et al. In addition, their work uses Tamarin [11] as its backend verification engine, which is different from the Horn clause based approaches (general comparison is difficult [21]). In general, Tamarin can be used for reasoning on protocols with global states, but its user may need to interact with the verifier [4]. Our verifier, on the other hand, is fully automatic. Furthermore, Mödersheim developed a verification framework that works with global states [22]. The framework extends the IF language with sets and abstracts messages based on its Set-Membership. However its expressivenss and verification applicability is unclear. Guttman extended the strand space with mutable states to deal with stateful protocols [23, 24], but without tool support for his approach. Most importantly, none of the above explicitly handles unbounded global state evolving.

6 Conclusions and Future Work

We have presented a new approach for the stateful security protocol verification with unbounded global state evolving. We implemented a tool for our new approach and the verification results of a number of protocols are quite encouraging. For future work, accelerating the redundancy checking would be helpful to improve the tool's performance. In addition, analyzing more stateful protocols, and adapting our approach for protocols with physical properties, e.g., time and space, would be interesting directions.

Acknowledgement. This work is supported by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No.NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

References

1. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Proc. 14th IEEE Computer Security Foundations Workshop (CSFW), IEEE CS (2001) 82–96
2. Viganò, L.: Automated security protocol analysis with the avispa tool. Electronic Notes in Theoretical Computer Science (ENTCS) **155** (2006) 61–86
3. Escobar, S., Meadows, C.A., Meseguer, J.: Maude-NPA: Cryptographic protocol analysis modulo equational properties. In: Foundations of Security Analysis and Design (FOSAD). Volume 5705 of LNCS., Springer (2009) 1–50

4. Kremer, S., Künnemann, R.: Automated analysis of security protocols with global state. In: Proc. 24th IEEE Symposium on Security and Privacy (S&P). (2014) 163–178
5. Herzog, J.: Applying protocol analysis to security device interfaces. IEEE Security & Privacy **4**(4) (2006) 84–87
6. Arapinis, M., Ritter, E., Ryan, M.D.: StatVerif: Verification of stateful processes. In: Proc. 24th IEEE Computer Security Foundations Symposium (CSF), IEEE CS (2011) 33–47
7. Delaune, S., Kremer, S., Ryan, M.D., Steel, G.: Formal analysis of protocols based on TPM state registers. In: Proc. 24th IEEE Computer Security Foundations Symposium (CSF), IEEE CS (2011) 66–80
8. Dong, N., Jonker, H., Pang, J.: Challenges in ehealth: From enabling to enforcing privacy. In: Proc. 1st International Symposium on Foundations of Health Informatics Engineering and Systems (FHIES). Volume 7151 of LNCS., Springer (2011) 195–206
9. Dong, N., Jonker, H., Pang, J.: Formal analysis of privacy in an ehealth protocol. In: Proc. 17th European Symposium on Research in Computer Security (ESORICS). Volume 7459 of LNCS., Springer (2012) 325–342
10. Dong, N., Jonker, H.L., Pang, J.: Formal modelling and analysis of receipt-free auction protocols in applied pi. Computers & Security **65** (2017) 405–432
11. Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN prover for the symbolic analysis of security protocols. In: Proc. 25th International Conference on Computer Aided Verification (CAV). Volume 8044 of LNCS., Springer (2013) 696–701
12. Li, L., Dong, N., Pang, J., Sun, J., Bai, G., Liu, Y., Dong, J.S.: A verification framework for stateful security protocols – full version (2017) Available online at `http://www.comp.nus.edu.sg/~dongnp/sspa`.
13. Ables, K., Ryan, M.D.: Escrowed data and the digital envelope. In: Proc. 3rd International Conference in Trust and Trustworthy Computing (TRUST). Volume 6101 of LNCS., Springer (2010) 246–256
14. Microsoft: Bitlocker FAQ (2011) Available online at `http://technet.microsoft.com/en-us/library/hh831507.aspx`.
15. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Communication of the ACM **21**(12) (1978) 993–999
16. Lowe, G.: An attack on the needham-schroeder public-key authentication protocol. Information Processing Letters **56** (1995) 131–133
17. Li, L., Dong, N., Pang, J., Sun, J., Bai, G., Liu, Y., Dong, J.S.: SSPA tool, experiment models and evaluation results (2017) Available online at `http://lilissun.github.io/r/sspa.html`.
18. Dolev, D., Yao, A.C.C.: On the security of public key protocols. IEEE Transactions on Information Theory **29**(2) (1983) 198–207
19. Rusinowitch, M., Turuani, M.: Protocol insecurity with a finite number of sessions, composed keys is np-complete. Theoretical Computer Science **299**(1-3) (2003) 451–475
20. Durgin, N.A., Lincoln, P., Mitchell, J.C.: Multiset rewriting and the complexity of bounded security protocols. Journal of Computer Security **12**(2) (2004) 247–311
21. Meier, S.: Advancing automated security protocol verification. PhD thesis, ETH (2013)
22. Mödersheim, S.: Abstraction by set-membership: verifying security protocols and web services with databases. In: Proc. 17th ACM Conference on Computer and Communications Security (CCS), ACM (2010) 351–360
23. Guttman, J.D.: Fair exchange in strand spaces. In: Proc. 7th International Workshop on Security Issues in Concurrency (SECCO). Volume 7 of EPTCS. (2009) 46–60
24. Guttman, J.D.: State and progress in strand spaces: Proving fair exchange. Journal of Automatic Reasoning **48**(2) (2012) 159–195