SmartVerif: Push the Limit of Automation Capability of Verifying Security Protocols by Dynamic Strategies

Yan Xiong, Cheng Su, Wenchao Huang, Fuyou Miao, Wansen Wang, and Hengyi Ouyang

School of Computer Science and Technology, University of Science and Technology of China

Abstract

Current formal approaches have been successfully used to find design flaws in many security protocols. However, it is still challenging to automatically analyze protocols due to their large or infinite state spaces. In this paper, we propose SmartVerif, a novel and general framework that pushes the limit of automation capability of state-of-the-art verification approaches. The primary technical contribution is the *dynamic* strategy inside SmartVerif, which can be used to smartly search proof paths. Different from the non-trivial and error-prone design of existing static strategies, the design of our dynamic strategy is simple and flexible: it can automatically optimize itself according to the security protocols without any human intervention. With the optimized strategy, SmartVerif can localize and prove *supporting lemmata*, which leads to higher probability of success in verification. The insight of designing the strategy is that the node representing a supporting lemma is on an incorrect proof path with lower probability, when a random strategy is given. Hence, we implement the strategy around the insight by introducing a reinforcement learning algorithm. We also propose several methods to deal with other technical problems in implementing SmartVerif. Experimental results show that SmartVerif can automatically verify all security protocols studied in this paper. The case studies also validate the efficiency of our dynamic strategy.

1 Introduction

Security protocols aim at providing secure communications on insecure networks by applying cryptographic primitives. However, the design of security protocols is particularly error-prone. Design flaws have been discovered for instance in the 5G [9], WiFi WPA2 [57], and TLS [23]. These findings have made the verification of security protocols a very active research area since the 1990s.

During the last 30 years, many research efforts [7,10,12,21–23,28,30,41] were spent on designing techniques to model and analyze protocols. The earliest protocol analysis tools, *e.g.*, the Interrogator [42] and the NRL Protocol Analyzer [40], could be used to verify security properties specified in temporal logic. Generic model checking tools have been used to analyze protocols, *e.g.*, FDR [39] and later Murphi [43]. More recently the focus has been on model checking tools developed specifically for security protocol analysis, such as Blanchet's ProVerif [12], the AVISPA tool [7], Maude-NPA [28] and tamarin prover [41]. There have also been hand proofs aimed at particular protocols. Delaune *et al.* [26] showed by a dedicated hand proof that for analyzing PKCS#11 one may bind the message size. Guttman [33] manually extended the space model by adding support for Wang's fair exchange protocol [58].

Unfortunately, although formal analysis has been successful in finding design flaws in many security protocols, it is still challenging for existing verification tools to support fully automated analysis of security protocols, especially protocols with global states [6, 32, 45, 48, 60] or unbounded sessions [15,47,49]. They may suffer non-termination during the verification mainly caused by the problem of state explosion. To avoid the explosion of the state space, several tools, *e.g.*, ProVerif [12] and AVISPA [7], use an abstraction on protocols, so that they support more protocols with unbounded sessions. Due to the abstraction, however, they may report false attacks when analyzing protocols with global states [6, 13]. StatVerif [6] and Set-π [13] extend the applied pi-calculus with global states, but the number of sessions they support is limited and they fail to automatically verify complicated protocols (*e.g.*, CANauth protocol [56]). GSVerif [17] enrichs ProVerif's proof strategy and supports several protocols with unbounded sessions [32, 48], but it fails to automatically verify complicated protocols (*e.g.*, Yubikey protocol [60]). Tamarin prover [36, 41] can verify more protocols without limitations of states or sessions, but it comes at the price of losing automation. It requires the user to supply insight into the problem by proving auxiliary lemmata, which is hard even for experts [36].

We propose and implement SmartVerif, a novel and gen-

eral framework of verifying security protocols. It pushes the limit of automation capability of state-of-the-art verification tools. Our work is motivated by the observation that these tools generally use a *static* strategy during verification, where design of the strategy is non-trivial. Here, the verification can be simply regarded as the process of path searching in a tree: each node represents a proof state which includes a lemma as a candidate used to prove the lemma in its parent and a path is correct if and only if each node on the path represents a *supporting lemma* which is a special lemma necessarily used for proving the specified security property. Therefore, the supporting lemmata have to be proven, before the property is verified.

Based on the observation, we design a *dynamic* strategy in SmartVerif. In other words, SmartVerif runs round-by-round, where in each round the strategy is either applied in searching until the complete proof path is selected, or optimized in case the current selected path is estimated incorrect. The initialization of the strategy does not need any human intervention, *i.e.*, the initial strategy is purely random. After the strategy is sufficiently optimized, it can smartly choose the next searching nodes. Specially, it efficiently localizes the node representing a supporting lemma among the nodes, which leads to success in verification. Recall that tamarin prover can let users supply supporting lemmata to reduce the complexity of automation. In comparison, the dynamic strategy in SmartVerif can help find the lemmata automatically and smartly, such that the protocols can be verified without any user interaction.

Our dynamic strategy builds upon the *insight* that the node representing a supporting lemma is on the incorrect path with lower probability, when a random strategy is given (See the proof in Appendix A). Hence, we introduce Deep Q Network (DQN) [44], a reinforcement learning agent, into the verification. The DQN updates the strategy according to historical incorrect paths. It uses an experience replay mechanism [38] which randomly samples previous transitions, and smooths the training distribution over the incorrect paths. As a result, an optimized strategy tends to select a node representing supporting lemma among the candidates, which leads to higher probability of successful verification.

We also propose to solve other technical problems in implementing SmartVerif. We present an approach of generating incomplete verification tree for reducing the memory overhead. We also design an algorithm of estimating correctness of selected paths to detect loops, which is the key component for supporting the DQN. Note that since we focus on the automation capability, we design SmartVerif based on tamarin prover that we modify tamarin prover for preprocessing protocol models and acquiring information for the DQN.

Experimental results show that SmartVerif can automatically verify all the studied protocols, without any human intervention. These protocols include Yubikey protocol [60] and CANauth protocol [56], which cannot be automatically verified by state-of-the-art verification tools. The case studies also validate the efficiency of our dynamic strategy.

The main contributions of the paper are three folds:

1. We present SmartVerif, to the best of our knowledge, the first framework that automatically verifies security protocols by dynamic strategies.

2. We propose several methods to deal with technical problems in implementing the framework. Specifically, we achieve our dynamic strategy by using the DQN and designing the rewards based on the insight. We design the algorithm of estimating the correctness of selected paths by detecting loops on the path. We propose to generate the incomplete verification tree to reduce the memory overhead. We implement a multi-threading process of path selection for better efficiency.

3. SmartVerif pushes the limit of automation capability of protocol verification, and it greatly outperforms state-of-the-art tools. SmartVerif achieves two goals: generality in designing heuristics and full automation in verification.

The rest of the paper is organized as follows. We review some related work and introduce tamarin that we use in Section 2 and Section 3, respectively. Then, we present the overview of SmartVerif in Section 4. In Section 5, we show an illustrative example of a security protocol. Afterwards, we solve the main problems in designing the Acquisition and Verification module in Section 6 and Section 7, respectively. We report our extensive experimental results and briefly overview the Yubikey and CANAuth protocol as case studies in Section 8. Finally, we present our future work and conclude the paper. We also illustrate and prove our insight in Appendix A. We present detailed description of the DQN in Appendix B.

2 Related Work

There are several typical model checking approaches that can deal with security protocols. ProVerif [12], one of the most efficient and widely used protocol analysis tools, relies on an abstraction that encodes protocols in Horn clauses. This abstraction is well suited for the monotonic knowledge of an attacker, which makes the tool efficient for verifying protocols with an unbounded number of protocol sessions [11, 35]. It is capable of proving reachability properties, correspondence assertions, and observational equivalence. Protocol analysis is considered with respect to an unbounded number of sessions and an unbounded message space. StatVerif [6] is an extension of ProVerif with support for explicit states. Its extension is carefully engineered to avoid many false attacks. It is used to automatically reason about protocols that manipulate global states. GSVerif [17] extends ProVerif to global states. It provides several sound transformations that cover private channels, cells, counters, and tables. It is efficient to verify protocols with global states.

Another verification approach that supports the verification of stateful protocols is the tamarin prover [53], [41]. Instead of abstraction techniques, it uses backward search and lemmata to cope with the infinite state spaces in verification. The benefit of tamarin and related tools is a great amount of flexibility in formalizing relationships between data that cannot be captured by a particular abstraction and resolution approach. It can handle protocols with global states [36], unbounded sessions [41], observational equivalence properties [10] and XOR [9] *etc.* However it comes at the price of losing automation, *i.e.*, the user has to supply insight into the problem by proving auxiliary lemmata for complex protocols. Tamarin has already been used for analyzing the Yubikey device [37], security APIs in PKCS#11 [26] and a protocol in TPM [25]. Using tamarin prover, researchers have discovered attacks for protocols such as V2X [59].

Overall, current approaches provide efficient ways in verifying security protocols. However, they commonly adopt a static strategy during verification, which may result in non-termination when verifying complicated security protocols. Encountering these cases, human experts are needed to analyze the reason of non-termination and supply hand proof.

At the same time, fast progress has been unfolding in machine learning applied to tasks that involve logical inference, such as knowledge base completion [55] and premise selection in the context of theorem proving [34]. Reinforcement learning in particular has proven to be a powerful tool for embedding semantic meaning and logical relationships into geometric spaces. These advances strongly suggest that reinforcement learning may have become mature to yield significant advances in many research areas, such as automated theorem proving. To the best of our knowledge, SmartVerif is the first work that applies AI techniques to the automated verification of security protocols.

3 Preliminaries of Tamarin Prover

We firstly introduce tamarin prover that we modify. The tamarin prover [41] is a powerful tool for the symbolic modeling and analysis of security protocols. It takes a protocol model as input, specifying the actions taken by protocol's participants (*e.g.*, the protocol initiator, the responder, and the trusted key server), a specification of the adversary, and a specification of the protocol's desired properties. Tamarin can then be used to automatically construct a proof that, when many instances of the protocol's participants are interleaved in parallel, together with the actions of the adversary, the protocol fulfills its specified properties.

Protocols and adversaries are specified using an expressive language based on multiset rewriting rules. These rules define a labeled transition system whose state consists of a symbolic representation of the adversary's knowledge, the messages on the network, information about freshly generated values, and the protocol's state. The adversary and the protocol interact by updating and generating network messages. Security properties are modeled as trace properties, checked against the traces of the transition system.

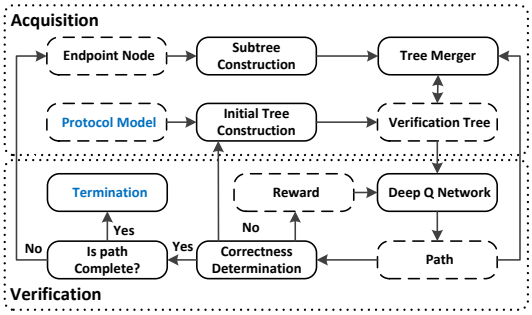


Figure 1: Framework of SmartVerif.

To verify a protocol, tamarin uses a constraint solving algorithm for determining whether $P \vDash_E \phi$ holds for a protocol P, a trace property ϕ and an equational theory E that formalizes the semantics of function symbols in protocol model. The verification always starts with either a simplification step, which looks for a counterexample to the property, or an induction step, which generates the necessary constraints to prove the property. On a high level, the algorithm can be regarded as the process of path searching in a tree. Each node of the tree represents an independent constraint system. Intuitively, tamarin applies constraint reduction rules to a constraint system to generate a finite set of refined systems. Note that tamarin prover uses these rules to represent lemmata in verification process. This problem is undecidable and the algorithm does not always terminate. Nevertheless, it often finds a counterexample (an attack) or succeeds in unbounded verification.

To search the solved constraint system, tamarin prover uses a heuristic to sort the applicable rules for a constraint system. The design rationale underlying tamarin's heuristic is that it prefers rules that are either trivial to solve or likely to result in a contradiction. Since the rules are sorted, tamarin always chooses the first rule to refine the current system, *i.e.*, expand only one endpoint node in the search tree. The rest endpoint nodes remain collapsed. Hence, the search tree is simplified to a finite one, which reduces the complexity of verification process.

4 Overview

Briefly, given a tamarin protocol model, *i.e.*, a protocol description and a security property, the workflow of SmartVerif consists of the following steps:

- Step 1: The Deep Q Network (DQN) is initialized with a purely random strategy, which takes multiple candidates as input, and randomly chooses a candidate with the uniform probability as output.
- Step 2: SmartVerif conducts proof searching by using the current strategy. It executes in parallel by multi-

threading. In each thread, a proof path is generated using the tamarin prover as backend, where each node on the path is chosen according to the strategy.

- Step 3: If a path generated in Step 2 is correct and complete, SmartVerif terminates and outputs the path as the result.
- Step 4: Otherwise, if all the paths generated in Step 2 are estimated incorrect, SmartVerif starts a new epoch where the DQN is trained according to the proof paths, and the strategy is updated. Here, we use the term epoch to denote the time step in which the DQN is optimized with new rewards.
- Step 5: Go to Step 2.

As shown in Figure 1, SmartVerif contains Acquisition and Verification module, which execute in multiple rounds:

Acquisition: The module generates a verification tree as input of Verification module. A path in the tree corresponds to a possible proof path in verification. Each node in the tree contains information for guiding the verification. We modify tamarin prover to collect the information. Since information of the nodes is transformed as input of the DQN, for which handling high-dimensional data is difficult, the information must be carefully chosen to reduce complexity of designing the DQN. Moreover, we face a problem of constructing the verification tree. There are protocols with large or infinite state spaces [48, 60]. In this case, even little information stored in nodes would still lead to memory explosion. To solve the problem, we design the DQN to guide the tree generation. Specifically, the tree is generated and expanded gradually that in each round only one of the endpoint nodes in the current tree is expanded, and the rest endpoint nodes remain collapsed for reducing the complexity of the tree. Here, the selection of the endpoint node is guided by the current strategy in DQN.

Verification: The module selects a path from the verification tree as a candidate proof. The path selection is guided by a dynamic strategy which uses the DQN. Meanwhile, the strategy is also optimized with correctness of the selection. Here, we additionally illustrate how the submodules in the Verification module deal with the tree. Briefly, there are 2 submodules.

1) Correctness Determination: It estimates whether the DQN selects the correct path. The main idea is to detect whether there are loops along the path (See Section 7.1). In each round, SmartVerif works according to the selected path in different cases:

Case 1: *The path is estimated incorrect*. We optimize the DQN in this epoch by passing rewards to the DQN. Meanwhile, we start a new epoch and send feedback to Acquisition module, where the submodule of Initial Tree Construction is informed to regenerate a new verification tree. As a result, we can find a new proof path according to the optimized DQN afterward.

Case 2: *The path is estimated correct but incomplete*. The incompleteness of the path is caused by the incompleteness of the verification tree. Therefore, we inform the submodule of Subtree Construction to expand the tree in the next round, so the path is also extended in the next round for shaping a complete path.

Case 3: *The path is correct and complete*. In this case, we achieve a successful verification of the protocol model, so we can terminate SmartVerif.

2) Deep Q Network: We introduce the DQN to update the dynamic strategy in SmartVerif. The key of the design of DQN is constructing the reward. In SmartVerif, the DQN selects a path from the verification tree in each round. Specifically, for each node that is on an estimated incorrect path, the node is bound to a negative reward. The design of the reward corresponds to our insight as mentioned in Section 1. This insight enables us to leverage the detected paths to guide the path selection.

5 Example

To illustrate our method, we consider a simple security protocol. The goal of the protocol is that when a participant C sends a symmetric key k to another participant S, the secret symmetric key k should not be obtained by the adversary.

$$S_1.\ C \to S: \qquad \{T_1,\ k\}_{pk_s}$$
$$S_2.\ S \to C: \qquad \{T_2,\ h(k)\}$$

Figure 2: A simple security protocol.

The brief process of the protocol is shown in Figure 2. In step S_1, C generates a symmetric key k, encrypts a tuple $\{T_1, k\}$ with the public key of S, and sends the encrypted message. Here, tag T_i is used to annotate protocol step i in protocol execution. In step S_2, S receives C's message, decrypts it with its private key, and gets the symmetric key k. Finally, S confirms the receipt of k by sending back its hash $h(k)$ to C.

The communication network is assumed to be completely controlled by an active Dolev-Yao style adversary [27]. In particular, the adversary may eavesdrop the public channels or send forged messages to participants according to the channels. Moreover, the adversary can access the long-term keys of compromised agents. Besides, the adversary is limited by the constraints of the cryptographic methods used. For example, it cannot infer hash input from hash output.

Here, we provide a brief explanation on modeling protocols in tamarin prover. A tamarin model defines a transition system whose state is a multiset of facts. The transitions are specified by rules. At a very high level, tamarin rules encode the behavior of participants and adversaries. Tamarin rules $[l] - [a] \to [r]$ have a left-hand side l (premises), actions a, and a right-hand side r (conclusions). The left-hand

and right-hand sides of rules respectively contain multisets of facts. Facts can be consumed (when occurring in premises) and produced (when occurring in conclusions). Each fact can be either linear or persistent (marked with an exclamation point !). While we use linear facts to model limited resources that cannot be consumed more times than they are produced, persistent facts are used to model resources which can be consumed any number of times once they have been produced. Actions are a special kind of facts. They do not influence the transitions, but represent specific states in protocol. These states form the relation between transition system and the security property.

Security properties are specified in a fragment of first-order logic. Tamarin offers the usual connectives (where & and | denote "and" and "or", respectively), quantifiers All and Ex, and timepoint ordering $<$. Note that the negation connective does not exist in the modeling language. Besides, while & and | have the similar meanings as in C-family programming languages, ! does not. In formulas, the prefix # denotes that the following variable is of type timepoint. Besides, tamarin offers two connectives @ and $\triangleright_0$ for stating the relations between facts and timepoints. For example, the expression $\mathbf{Action}(args)@\#t$ denotes that $\mathbf{Action}(args)$ is executed at timepoint $\#t$. The expression $\mathbf{Action}(args) \triangleright_0 \#t$ denotes that $\mathbf{Action}(args)$ is executed before timepoint $\#t$.

For instance, to model the above protocol, we first define several functions and predicates. 1) $\mathbf{In}(m)$ and $\mathbf{Out}(m)$: message m is sent and received, respectively; 2) $\mathbf{aenc}\{a\}k$ and $\mathbf{adec}\{a\}k$: asymmetric encryption and decryption of a variable a using key k; 3) $\mathbf{Pk}(A,\ pk_A)$ and $\mathbf{Ltk}(A,\ ltk_A)$: participant A is bound to a public key pk_A and a private key ltk_A, respectively; 4) $\mathbf{fst}\{a,\ b\}$ and $\mathbf{snd}\{a,\ b\}$: the first and second element from a tuple $\{a,\ b\}$, respectively; 5) $\mathbf{Eq}(a,\ b)$: a is equal to b; 6) $\mathbf{h}(a)$: the result of hashing a.

Then, the compromise of private keys is modeled using the following rule.

rule *Reveal_ltk* :
[!**Ltk**(A, ltk_A)] – [**LtkReveal**(A)] → [**Out**(ltk_A)]

It has a premise $!\mathbf{Ltk}(A,\ ltk_A)$ which binds the private key ltk_A to a participant A. The corresponding conclusion $\mathbf{Out}(ltk_A)$ states that the private key ltk_A is sent to the adversary. Note that, this rule has an action $\mathbf{LtkReveal}(A)$ stating that the key of A was compromised. This action is used to model the security property.

Then, the protocol is modeled using the rules in Figure 3. Rule *C_1* captures a participant generating a fresh key and sending the encrypted message. The rule has two facts for premises. The first fact $\mathbf{Fr}(k)$ states that a fresh variable k is generated. The second fact $!\mathbf{Pk}(S,\ pk_S)$ states that the public key pk_S is bound to a participant S. In this case, the second fact is a persistent fact since the public key can be used in many times (*i.e.*, by protocol participants or adversaries). If the facts in premises are matched with the facts in the current

rule *C_1* :
[**Fr**(k), !**Pk**(S, pk_S)] – [ ] → [**Send**(S, k), **Out**(**aenc**$\{T_1,\ k\}pk_S$)]

rule *S_1* :
[!**Ltk**(S, ltk_S), **In**(*request*)] – [**Eq**(**fst**(**adec**(*request*, ltk_S)), T_1)]
→ [**Out**(T_2,**h**(**snd**(**adec**(*request*, ltk_S))))]

rule *C_2* :
[**Send**(S, k), **In**(**h**(k))] – [**SessKeyC**(S, k)] → [ ]

Figure 3: The model of the protocol process.

state, two conclusions are produced. The first is an action $\mathbf{Send}(S,\ k)$ which states that k is sent to a participant S. The second conclusion is $\mathbf{Out}(\mathbf{aenc}\{T_1,\ k\}pk_S)$. This fact states that the participant uses a public key pk_S to encrypt the message $\{T_1,\ k\}$ and send the message. Rule *S_1* captures a participant receiving the message sent by C and sending the hash value of k back. Rule *C_2* captures a participant receiving the hash value and completing a run of the protocol.

Finally, we define a security property, which states that when a participant C sends a symmetric key k to another participant S , the secret symmetric key k should not be obtained by the adversary. The security property is modeled as a lemma Key_secrecy in Figure 4. The lemma indicates that, there must not exist a state, where action $\mathbf{SessKeyC}(S,\ k)$ happens and the adversary obtains k, without the happening of the compromise action $\mathbf{LtkReveal}(S)$.

lemma *Key_secrecy* :
$''not(\ Ex\ S\ k\ \#i\ \#j.\ SessKeyC(S,\ k)\ @\ \#i\ \&\ K(k)\ @\ \#j$
$\&\ not(Ex\ \#r.\ LtkReveal(S)\ @\ r)\ )''$

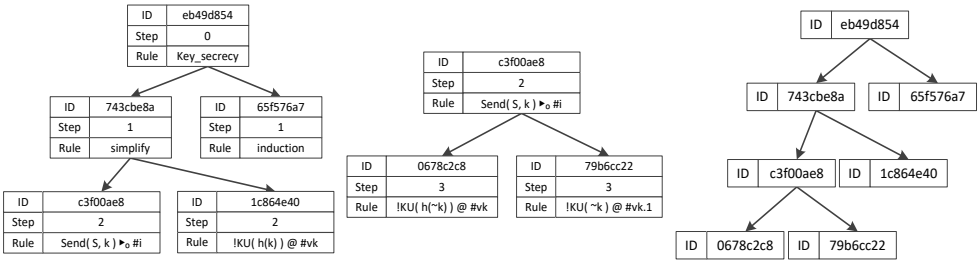
Figure 4: The security property.

Note that the above security property of protocol can be successfully verified by tamarin prover. To better understand the following sections, we use this protocol as an example. We describe how we generate the verification tree of the protocol in Section 6. Then we explain how SmartVerif verifies protocols in Section 7.

6 Acquisition module

6.1 Choosing Information

The information in nodes of the verification tree is used in 2 ways. 1) We transform the information to input of the DQN. In the Verification module, we use the DQN to select a proof path in verification tree. The DQN in Verification module requires an input state, which represents current proof state. We use the information to represent proof state in verification process. Since it is difficult for the network to handle high-dimensional data, the input of the network should not be large in dimensions. Hence, we do not choose all the intermediate



(a) Verification tree (b) Subtree (c) Merged tree

Figure 5: Construction of a Verification Tree.

data in the verification process as the information. 2) We use the information to distinguish different proof states. Note that SmartVerif runs round-by-round, where in each round verification trees are constructed and merged. In the merging process, we compare the information in nodes in different trees to find a same proof state in each round. Therefore, the information in the node should not only be simple enough, but also represent independent state in the verification process.

For each node in the verification tree, we choose the constraint reduction rule and and step number, *i.e.*, distance from the node to the root, as the stored information. Recall that at each proof step tamarin prover applies a constraint reduction rule to refine a constraint system. Hence, rules and their step numbers can represent independent states in the verification process. As illustrated in Figure 5, each node in the tree contains three pieces of information as follows: 1) ID: the hash value of the constraint reduction rule; 2) Step: the current proof step number; 3) Rule: the string of the constraint reduction rule. Here, the hash value is shown with the first eight characters for abbreviation.

Considering the protocol in Section 5, we show the information collected from modified tamarin prover in Figure 5 (a). Due to the limitation of paper size, we only demonstrate the information collected in the first two steps. Specifically, the root node $eb49d854$ represents the lemma *Key_secrecy*. In the first proof step, tamarin prover constructs the proof starting with either an *induction* rule, which generates the necessary constraints to prove the lemma, or a *simplify* rule, which generates initial constraint system to look for a counterexample to the lemma. In this case, tamarin prover chooses rule *simplify* at proof step #1, which corresponds to node $743cbe8a$. In detail, it looks for a protocol execution that contains a $\mathbf{SessKeyC}(S,\ k)$ and a $\mathbf{K}(k)$ action, but does not use an $\mathbf{LtkReveal}(S)$. As shown in Figure 3, action $\mathbf{SessKeyC}(S,\ k)$ is in the protocol execution only if the protocol step that rule C_2 captures has happened. Since rule C_2 has two premises $\mathbf{Send}(S,\ k)$ and $\mathbf{In}(\mathbf{h}(k))$, these two facts are in the protocol execution. Based on this observation, tamarin prover has two constraint reduction rules to select: $Send(S,\ k) \triangleright_0 \#i$ and $KU(h(k))@\#vk$. The first rule $Send(S,\ k) \triangleright_0 \#i$ states that action $\mathbf{Send}(S,\ k)$ is executed before timepoint $\#i$. The second rule $KU(h(k))@\#vk$ states that the adversary knows k's hash value at timepoint $\#vk$. Therefore, in the tree, node $743cbe8a$ has two children which represent these two rules respectively.

6.2 Tree Construction

We construct a verification tree to store the information we collect. The tree is used in Verification module to generate a candidate proof by guidance of the DQN. As illustrated in Section 4, to avoid memory explosion, we design a simple and effective approach. We firstly initialize a tree with a root starting from the security property. Each node in the tree contains information specified in Section 6.1. Then, in each new round, when the tree is expanded, an endpoint node in the current tree is chosen according to the DQN, and a depth-two subtree is generated. The root of the subtree is the chosen endpoint node and the nodes of the second depth represent the possible constraint reduction rules that can be used to prove the lemma of the root. Therefore, the new tree is formed by merging the subtree into the current tree.

In Figure 5, we exemplify the construction of the verification tree for the protocol in Section 5. In the initial round, the Acquisition module generates a tree, whose root node represents the lemma *Key_secrecy*, as shown in Figure 5(a). In the next round, if the DQN in the Verification module selects the endpoint node $c3f00ae8$ as the estimated supporting lemma, the Acquisition module uses the modified tamarin prover to go one step further, gathers the information, and constructs a subtree shown in Figure 5(b). Then, the acquisition merges the subtree into the current tree as shown in Figure 5(c).

Besides, we implement a multi-threading process of path selection for better efficiency. Recall that the DQN optimizes itself with its selection and the corresponding rewards. Since it selects only one path given a verification tree at each round, the quantity of training data is limited, which decreases the training efficiency and lowers the performance. To solve this problem, we execute multiple threads of the Acquisition module in parallel to generate various verification trees for the DQN. Therefore, the DQN selects multiple paths in these trees at a time and generate more training data for optimizing. Using this approach, we are able to achieve greater data efficiency and increase the convergence rate. We further validate and evaluate the multi-threading process in the experiments in Section 8.1.

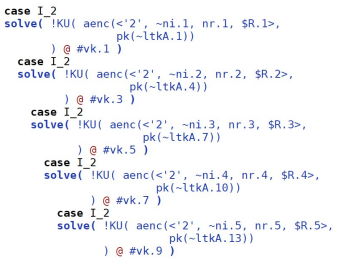


Figure 6: An example of verification loop.

7 Verification module

Briefly, the Verification module selects a proof path from the verification tree. The selection is guided by our dynamic strategy and an algorithm of correctness determination of the selection. In Section 7.1, we describe our method of correctness determination. We describe the design of DQN in Section 7.2. We then analyze the DQN in Section 7.3.

7.1 Correctness Determination

We illustrate how we determine the correctness of a proof path. The main idea is to detect whether there are loops along the path. For example, Figure 6 shows the information of the last 5 consecutive nodes on a path, when using tamarin and encountering the loop on the path. Here, words in blue indicate constraint reduction rules selected by tamarin prover, and words in black indicate tags marked by tamarin prover in verification. We find that the rules of the nodes are similar, which has the following form:

$$\mathsf{!KU}(\mathsf{aenc}(<'2',\ \mathsf{ni.1}, \mathsf{nr.1}, \$\mathsf{R.1}>, \mathsf{pk}(\ \mathsf{ltkA.1}))) \ @\ \#\mathsf{vk.1})$$

Therefore, we consider the path incorrect since the loop on it results in a non-termination in searching. Besides, there are several other kinds of loops. For example, the sequence of the nodes may have the form $[\ldots,a,b,a,b]$ or $[\ldots,a,b,c,a,b,c]$, where a,b,c are constraint reduction rules. The loops on these sequences may also lead to a failed verification. Hence, the loop detection algorithm should be carefully designed to determine the correctness of the path.

Based on the observation, we design the algorithm of loop detection as shown in Algorithm 1. The algorithm takes a string sequence $[s_1,s_2,...,s_k]$ as input. The sequence is transformed from the selected path. Each element, *e.g.*, s_i, is the constraint reduction rule of the corresponding node, *i.e.*, the ith node, on the path. The algorithm runs iteratively by generating a sequence S (line 4), and counting the number of pairs of similar elements that are both in S. If the number is not less than δ, a loop is detected. For example, given the aforementioned sequence $[\ldots,a,b,c,a,b,c]$, $S=(c,c,\ldots)$, when $j=3$. Note that the length of S is restricted to the limit of ρ for efficiency. For the same reason, the algorithm is activated only when the length of the selected path reaches α (line 1).

Algorithm 1 Loop Detection Algorithm

Require: $(s_1,s_2,...,s_k)$
1: **if** $k \geq \alpha$ **then**
2: **for** $j=1$ to $k-1$ **do**
3: $count = 0$
4: $S = (s_k, s_{k-j}, s_{k-2j}, \ldots)$ //$|S| \leq \rho$
5: **for** each element x in S **do**
6: **for** each element y in S **do**
7: **if** $x \neq y$ **then**
8: n = length of x, m = length of y
9: **for** $a=1$ to n **do**
10: **for** $b=1$ to m **do**
11: **if** $y[a] == x[b]$ **then** $cost=0$ **else** $cost=1$
12: $v[b] = \min(v[b-1]+1, b, b-2+cost)$
13: **if** $v[m]/m < \beta$ **then** $count = count+1$
14: **if** $count \geq \delta$ **then return** TRUE
15: **return** FALSE
16: **else return** FALSE

We use Levenshtein distance to measure the similarity between two rules (line 8 to 13). The distance between two strings x and y is the minimum number of single-character edits (insertions, deletions or substitutions) required to change x into y. If the distance between x and y is less than β of the length of x, we assume these two strings are similar. For example in Figure 6, the difference points among the similar rules are often the numbers at corresponding positions, *e.g.*, $ni.1, ni.2, ni.3$. Informally, the number of difference points are counted as the distance.

Note that there is not necessarily a loop on every incorrect path. In other words, if a loop is found given a selected path, there may be multiple **incorrect nodes**, *i.e.*, the nodes that do not represent supporting lemmata, on the path. Therefore, it is not sufficient to use naive search algorithms, *e.g.*, DFS, to locate proof paths. We make further studies on analyzing the effectiveness of the algorithms in Section 8. Finally, in our implementation, we set α to 20, β to 0.1, ρ to 20 and δ to 3.

7.2 Deep Q Network

To apply our insight to SmartVerif, we introduce Deep Q Network (DQN) into the verification. Briefly, the DQN in SmartVerif maintains an action-selection policy which takes a state s_t as input, and outputs an action number a_t. Here, s_t represents the proof state, *i.e.*, a node in the verification tree, and a_t is the number of a chosen child of the node. The state s_t is a vector transformed from information in nodes, *e.g.*, the constraint reduction rule. In each epoch, the DQN uses a dynamic strategy to select paths in the verification tree. When the paths are estimated incorrect, the DQN optimizes its current strategy by training.

Specifically, given a group of estimated incorrect paths, the training process works as follows. *First*, for each node that is on a path of the group, the node is bound to a negative reward. The reward is related to the probability that the node represents a supporting lemma, according to our insight. In

the implementation, a tuple, which includes the node and the reward, is added into a global dataset. *Then*, a subset is sampled from the dataset. *Finally*, the parameters of the DQN is optimized by minimizing a loss function. Here, the loss function is the sum of sub-functions. Each sub-function takes an individual tuple in the subset as input, and outputs the difference between results calculated according to two predefined functions. One of the functions is calculated by using the parameters of current DQN, and the other is calculated by using the training parameters of the optimized DQN. Informally, the training process optimizes the network in order that the reward of each node can be estimated. In other words, the node with the highest reward among its siblings can be regarded as the one with supporting lemma, if the DQN is sufficiently optimized. We demonstrate the technical details of our implementation of DQN in Appendix B.

7.3 Analysis of our DQN

The advantage of applying DQN is that DQN can update our dynamic strategy efficiently if the rewards in DQN are designed effectively. In other words, sufficient tuples are required to be updated in the dataset in each epoch, and the reward in each tuple should be correct. A naive algorithm is that only the reward corresponding to the last node on the incorrect path is set negative. However, the number of updated tuples is limited. Instead, we significantly improve the efficiency by storing multiple tuples as illustrated, and the correctness of the insight is analyzed as follows.

Recall that the reward in our DQN is related to the probability that the node represents a supporting lemma. Formally, we use $[n_{t_1}, n_{t_2}, ..., n_{t_{k_t}}]$ to denote a proof path t, where n_{t_i} is the ith node in the path and k_t is the number of nodes in the path. Therefore, the lemmata in $\{n_{t_i}\}$ are the candidate lemmata. For a node n_{t_i}, we use x_{t_i} and y_{t_i} to denote its degree and the number of its children which represents supporting lemmata respectively. Suppose there are at least one child of n_{t_i} that does not represent supporting lemma, *i.e.*, $x_{t_i} > y_{t_i}$. The random strategy here means that whenever choosing a child for searching, the probability of choosing is uniform. In other words, the probability of choosing any child of n_{t_i} is $\frac{1}{x_{t_i}}$. Hence, if the random strategy is applied in choosing a child of n_{t_i}, the probability of choosing the node representing supporting lemma is $\frac{y_{t_i}}{x_{t_i}}$. Suppose there are R correct and complete proof paths in a given tree.

Theorem. *Given the above assumptions, if n_{t_i} has been chosen, the node representing a supporting lemma, who is the child of n_{t_i}, is on an incorrect path with the probability less than $\frac{y}{x}$, when the random strategy is given.*

Proof. See Appendix A. □

To illustrate the theorem, we briefly study a naive and seemly good algorithm, which traverses the verification tree by DFS, selects nodes using existing static strategy when traversing, and backtracks according to our algorithm of correctness determination.

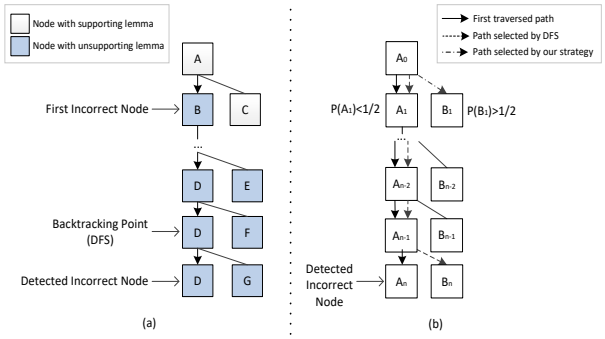


Figure 7: Verification trees: (a) A non-zero distance between the first incorrect node and the detected incorrect node. (b) The probability of nodes representing supporting lemmata after the first traversing.

1) Insufficiency of only applying correctness determination: The naive algorithm may have to explore a large amount of nodes. For example in Figure 7 (a), since the algorithm of correctness determination cannot detect all incorrect nodes, there is a non-zero distance between the first incorrect node B and the first detected incorrect node D. Hence, the naive algorithm has to traverse all paths that pass through B, before it traverses the correct path. As a result, the number of explored nodes grows exponentially with the distance.

2) Advantages of our DQN: Our DQN greatly outperforms the naive algorithm according to the theorem. By optimizing itself in each epoch, our DQN adjusts the Q value of each node in the tree. In this case, it tends to select the nodes with higher Q values as "good nodes", which have higher probability of representing supporting lemmata. For example in Figure 7 (b), assume that our DQN and the naive algorithm traverses along the same path $[A_0, A_1, ..., A_n]$, until the algorithm of correctness determination detects the incorrect node for the first time. Then, the naive algorithm backtracks from A_n, and continues traversing along $[A_0, A_1, ..., B_n]$. Assume that A_0 has two children. According to our theorem, the probability of A_1 being a supporting lemma become less than $\frac{1}{2}$, *i.e.*, $P(A_1) < \frac{1}{2}$, which was $\frac{1}{2}$ before the A_n is identified as an incorrect node. Similarly, the probability of all $A_1, A_2, \ldots, A_{n-1}$ being supporting lemmata decreases exponentially. Hence, B_n should not be the next traversed node due to the low probability. In comparison, our DQN sets the rewards of nodes $A_1, A_2, \ldots, A_n$ as negative values, such that the updated strategy in DQN tends to select B_1 instead of A_1. The analysis is also validated in our experiments (See Section 8.1).

8 Experiments & Evaluation

We perform experiments on several security protocols. The experiment results are described in Section 8.1. In Section 8.2,

we briefly overview Yubikey and CANAuth protocol to validate the efficiency of SmartVerif. All files of our prototype implementation and protocol models used in benchmark are available here [3].

8.1 Main Experiments

We compare SmartVerif with other verification tools in verifying security protocols. We evaluate the efficiency of SmartVerif.

Experimental Setup: Experiments are carried out on a server with Intel Broadwell E5-2660V4 2.0GHz CPU, 128GB memory and four GTX 1080 Ti graphic cards running Ubuntu 16.04 LTS. We use and modify tamarin prover v1.4.0 in SmartVerif.

We use the same network architecture, learning algorithm and parameter settings across all chosen protocols. Since the security property varies greatly in protocols, we set all the negative rewards to -10 for generality. In these experiments, we use the DQN with 0.01 learning rate and memory batch of size 7000. Moreover, we execute eight threads of Acquisition module in parallel. The behavior policy during training was ε-greedy with ε annealed linearly from 0.99 to 0.1 over the first hundred epochs, and fixed at 0.1 thereafter.

Chosen Tools: For each protocol with unbounded sessions, we inspect whether it can be automatically verified by SmartVerif and other verification tools. These verification tools include StatVerif [6], Set-π [13], tamarin prover [36, 41] and GSVerif [17]. The tools are typical verification tools which support verification of security protocols with global states. Moreover, all these tools provide automated verification modes to verify security protocols. Note that we attempt to verify protocols in several versions of tamarin prover. We first attempt to use the 's' heuristic of tamarin prover. The heuristic is the default heuristic of tamarin. We then attempt to use the consecutive heuristic (-'c') heuristic of tamarin prover. This heuristic adopts a simplest method to verify protocols: it solves goals in the order they occur in the constraint system. Unlike other default static strategies, this method does not contain any human-designed heuristics or expertise. We compare SmartVerif with this mode of tamarin prover to demonstrate the generality of our framework. Then, we try to verify protocols using the dedicated heuristic (-'p'), which is designed by the SAPIC authors [36] to efficiently solve SAPIC generated Tamarin models. Since we use tamarin protocol models as well as SAPIC generated models in our experiment, we additionally compare SmartVerif with this heuristic to validate the efficiency of our framework. Moreover, we implement two naive algorithms (DFS and BFS) with random strategy and our loop detection method. We further compare these two with our algorithm to show the efficiency of SmartVerif. Besides, we combine the DFS with the heuristics of tamarin prover to further validate the efficiency. Note that we do not choose classical verification tools, *i.e.*, ProVerif and AVISPA, since their support for protocols with global states and unbounded sessions is limited.

Chosen Protocols: We carefully choose security protocols to be testified in our evaluation.

1) We choose all the protocols that have been evaluated in papers of the compared tools, *i.e.*, StatVerif [6], Set-π [13], GSVerif [17], tamarin prover [41], SAPIC [36]. The chosen protocols include a simple security API similar to PKCS#11 [48], the Yubikey security token [60], the optimistic contract signing protocol by Garay, Jakobsson and MacKenzie (GJM) [32], *etc.* These protocols are typical protocols with global states, unbounded sessions. Many research efforts [6, 13, 36, 41] were spent on verification of these protocols. Besides, GSVerif paper evaluated the performance of 18 protocols, which are all chosen in our paper. In these protocols, Yubikey is the most important case for evaluation for it is most widely studied, but still have not been automatically verified, according to the current literature [13, 17, 37].

2) Since the security property of observation equivalence [16, 24, 29, 31, 52] cannot be verified by StatVerif, Set-π, or GSVerif while only tamarin provers supports verifying the property, we choose 5 protocols with the properties from the official repository [4] of tamarin. Specifically, these protocols include Chaum's Online e-Cash [16], FOO Voting [31], Denning-Sacco [24], Okamoto [52], and Feldhofer's RFID protocol [29].

3) For fairness we do not choose other protocols. Practically there are many other practical protocols that cannot be automatically verified by state-of-the-art tools, *e.g.*, TLS [23], 5G AKA [9] , smart contract and blockchain protocols [1, 2]. Note that supporting lemmata have to be the manually specified to help prove TLS [22] in tamarin prover. In comparison, SmartVerif successfully verifies these protocols, *e.g.*, TLS 1.3 [5] (totally 7 hours for all the properties, without using any human-written lemma). However, we do not compare SmartVerif with existing tools by using the protocols, since it becomes questionable whether the protocols are cherry-picked and whether some of the protocols can be verified by customized heuristics, *e.g.*, three protocols verified by optimized heuristics of GSVerif in Table [17]. Instead, since the protocols evaluated in the papers [6, 13, 17, 36, 41] are thoroughly studied, the experimental results on the protocols are more convincing.

Note that, for each chosen protocol, we only analyze the verified security properties in our experiments. There exists security properties which are falsified in the chosen protocols (*e.g.*, Denning-Sacco protocol [24]). Since the quantity of the proof steps of a falsified property is smaller than the quantity of the proof steps of the property after the protocol model is corrected, we did not analyze the falsified properties in our experiments.

Comparative Results: The experimental results are summarized in Table 1. Compared with these verification tools, SmartVerif is sufficient for generality and automation capabil-

Table 1: Experimental results on security protocols with unbounded sessions in verification tools.

Protocols	StatVerif	Set-π	GSVerif	SAPIC/Tamarin Prover													SmartVerif							
				-'s'			-'c'			-'p'			DFS		BFS		Automated Verification?	Training Epochs	QN	Training Time			Verification Time	Total Time
				Original		w/ DFS	Original		w/ DFS	Original		w/ DFS								Acquisition	Network Training	Overall		
				QN	Time		QN	Time		QN	Time		QN	Total Time	QN	Total Time								
Yubikey [60]	×	×	×	N/A	×	×	N/A	×	×	N/A	×	×	N/A	×	N/A	×	✓	79	9357	19m	40m	59m	2m	61m
Mödersheim's Example [45]	×	7s	6s	N/A	×	×	N/A	×	1h	11	1s	5s	4593	1h	31414	2h	✓	19	3345	6m	8m	14m	1m	29m
Security API in PKCS#11 [48]	×	×	15s	398	71s	83s	473	197s	225s	419	67s	79s	N/A	×	N/A	×	✓	175	12461	31m	50m	81m	2m	83m
GJM Contract-Signing [32]	10s	×	7s	27	19s	28s	33	64s	79s	37	27s	38s	6134	1h	33319	2h	✓	16	2579	3m	9m	12m	1m	13m
one-dec [17]	×	×	9s	N/A	×	7h	N/A	×	10h	N/A	×	10h	128013	6h	N/A	×	✓	31	3651	6m	16m	22m	1m	23m
one-dec, table variant [17]	×	×	8s	470	11s	23s	470	21s	36s	172	5s	13s	55891	4h	155891	5h	✓	25	3247	7m	13m	20m	1m	21m
private-channel [17]	×	×	7s	10	2s	4s	10	7s	19s	10	2s	9s	3149	1h	34390	2h	✓	28	2931	5m	10m	15m	1m	16m
counter [17]	×	×	9s	39	11s	15s	85	26s	40s	39	12s	23s	4301	1h	32314	2h	✓	29	3045	6m	12m	18m	1m	19m
voting [17]	×	×	7s	55	1s	3s	30	7s	16s	55	1s	8s	156931	5h	N/A	×	✓	31	5524	9m	16m	25m	1m	26m
TPM-envelope [25]	×	×	□	N/A	×	5h	N/A	×	9h	N/A	×	8h	36501	6h	206903	7h	✓	33	4583	7m	15m	22m	1m	23m
TPM-bitlocker [25]	5s	6s	□	N/A	×	1h	N/A	×	1h	N/A	×	1h	4392	2h	33391	1h	✓	36	3249	5m	9m	14m	1m	15m
TPM-toy [25]	×	×	□	N/A	×	1h	N/A	×	1h	N/A	×	1h	4831	1h	23901	1h	✓	21	2592	4m	9m	13m	1m	14m
Key registration [13]	4s	6s	7s	108	9s	14s	176	18s	31s	108	10s	23s	4190	1h	43903	1h	✓	14	3143	3m	7m	10m	1m	11m
Secure device [6]	×	6s	7s	N/A	×	×	N/A	×	×	94	16s	35s	86831	6h	N/A	×	✓	33	4036	6m	11m	17m	1m	18m
MaCAN [14]	×	11s	14s	121	13s	25s	175	29s	41s	142	18s	36s	4813	1h	37731	2h	✓	32	3641	7m	17m	24m	1m	25m
CANauth [56]	×	×	×	N/A	×	1h	N/A	×	3h	N/A	×	2h	6383	3h	N/A	×	✓	32	3342	3m	11m	14m	2m	16m
CANauth simplified [56]	×	×	×	N/A	×	1h	N/A	×	2h	N/A	×	1h	4341	1h	N/A	×	✓	38	3257	4m	11m	15m	1m	16m
Mobile EMV [19]	×	×	6s	N/A	×	×	N/A	×	×	N/A	×	14h	239783	7h	N/A	×	✓	40	6156	12m	22m	34m	2m	36m
Scytl Voting System [20]	×	×	5s	8	36s	53s	9	44s	63s	8	34s	47s	277313	9h	N/A	×	✓	43	7142	16m	33m	49m	1m	50m
Chaum's Online e-Cash [16]	×	×	×	18	12s	25s	32	20s	34s	18	14s	25s	3931	1h	30183	2h	✓	19	3097	7m	13m	20m	1m	21m
FOO Voting [31]	×	×	×	64	7s	19s	308	19s	32s	78	3s	8s	4984	1h	40139	2h	✓	26	4176	8m	17m	25m	1m	26m
Feldhofer's RFID [29]	×	×	×	96	8s	21s	192	15s	27s	96	9s	13s	4644	1h	39390	2h	✓	36	5753	10m	22m	32m	1m	33m
Denning-Sacco [24]	×	×	×	12	1s	6s	12	6s	14s	12	1s	7s	3741	1h	33901	2h	✓	27	5124	8m	18m	26m	1m	27m
Okamoto [52]	×	×	×	34	1s	5s	217	17s	25s	34	1s	7s	N/A	×	N/A	×	✓	141	10931	20m	45m	65m	1m	66m

×: no automatic verification (computation time >48h, memory used >128GB) ✓: automatic verification
□: requiring optimizing heuristics to achieve automatic verification

ity: it is able to verify all the given protocols with unbounded sessions, without any human intervention.

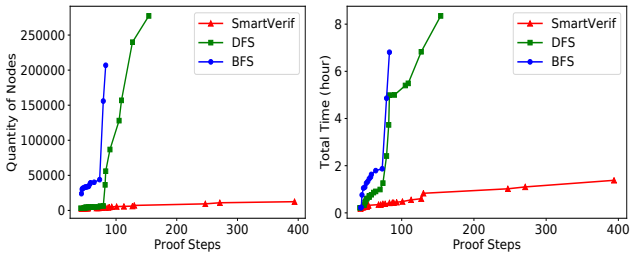
1) *Generality*. SmartVerif achieves a 100 percent success rate in verifying the studied protocols, which outperforms all the other verification tools. For example, StatVerif does not terminate and the verification fails encountering complicated protocols like security API in PKCS #11 and mobile EMV protocol [19]. Set-π fails in verifying TPM-envelope protocol [25] and some others with unbounded sessions. Tamarin prover is effective in automatically verifying simple protocols with unbounded sessions, *e.g.*, GJM Contract-Signing protocol and Security API in PKCS#11. Nevertheless, it does not achieve automated verification of Yubikey protocol, TPM-bitlocker, *etc.*, in any of the studied versions. GSVerif outperforms the previous tools in generality but it still can not automatically verify complicated protocols such as Yubikey protocol. Note that its heuristics in 3 cases are rewritten to achieve automation [17], *i.e.*, 11 cases cannot be automatically verified without the optimization. In comparison, our heuristic is designed without any human intervention. Moreover, among the tools, only tamarin prover and SmartVerif can handle protocols with observational equivalence properties. They achieve successful verification of the five protocol cases.

2) *Automation capability*. Currently, only SmartVerif can fully automatically verify Yubikey and CANauth protocol [56]. For protocols which existing tools cannot automatically verify, GSVerif and tamarin prover provide an interactive mode for users to manually guide the verification. In our experiments, we find that Yubikey and CANauth protocol can be verified by manually designing proof formulas using these tools. In contrast, SmartVerif fully automatically verifies these protocols, without any human intervention. In Section 8.2, we briefly overview Yubikey protocol to demonstrate the sufficiency of SmartVerif in automation capability. We further overview CANAuth protocol in Section 8.2.2.

Efficiency and Overhead: To evaluate the efficiency and overhead of SmartVerif, we collect statistics of the running time and training epochs in verification. The running time contains two parts: 1) *Training time*: the time spent in information acquisition and network training; 2) *Verification time*: the time spent in verification after network convergence. As presented in this paper, SmartVerif is a novel and general framework to verify protocols. For each protocol to be verified, it takes time to acquire information and train the network. Once the DQN is sufficiently optimized according to the current protocol model, it can be directly used to verify the corresponding protocol, like the static strategies in existing tools. Hence, we use the verification time to demonstrate the performance of SmartVerif. Besides, since SmartVerif uses the DQN to select proof paths, the efficiency of the DQN directly affects the performance and overhead of SmartVerif. Recall that the number of epochs denotes the times that the DQN is optimized with a new reward, which is related to the number of generated incorrect paths, if the protocol has not been successfully verified. Therefore, we use the quantities of training epochs and time of DQN to evaluate the efficiency of our framework.

As demonstrated in Table 1, the experimental results show that SmartVerif verifies the studied protocols in a very efficient way. For most protocols, it succeeds in verification only after about 25 times of one-way forward traversing (*i.e.*, 25 epochs). As the challenge is time explosion when traversing infinite state spaces, our dynamic strategy solves the problem in a general and adequate way. For instance, existing verification tools can not automatically verify Yubikey protocol with unbounded sessions due to memory explosion or infinite verification loops. In contrast, SmartVerif only takes 79 epochs to find the correct proof path using the dynamic strategy.

Moreover, the statistics of the running time also validate the efficiency of SmartVerif. Comparing to existing verification tools, SmartVerif does not require any extra time and effort in training human for interactive proving or designing heuristics.



(a) QN (b) Total time

Figure 8: Experimental results. (a): Quantity of nodes necessarily to be traversed. (b): Total time compared with naive algorithms.

Instead, it spends the training time on optimizing the dynamic strategy for protocols, which is sufficient. Specifically, if a protocol's model is complicated, *i.e.*, the searching space is large, the running time increases. The space's size depends on whether the model covers global states or unbounded sessions [48, 56, 60], or whether the model is simplified [22, 36]. For most protocols, it only takes less than half an hour to find the correct proof path. In the worst case, it costs 83 minutes to verify the security API in PKCS #11. After the DQN is sufficiently optimized, the verification time is only 2 minutes.

Performance Analysis: Besides proving our insight theoretically, we also perform empirical analysis by comparing SmartVerif with two naive algorithms: 1) **DFS**. The algorithm searches along a path as long as possible before backtracking. The backtracking occurs only when a loop is detected. 2) **BFS**. The algorithm searches all the paths at the present depth prior to searching at the next depth level. It also uses the loop detection algorithm to shrink the size of searching space. The BFS is optimized by multi-threading that each threads searches in parallel. Note that DFS has to run in a single thread since the ordering and parallel tends to conflict in searching. Both DFS and BFS are implemented based on tamarin prover and use the random strategy as the strategy of selecting nodes.

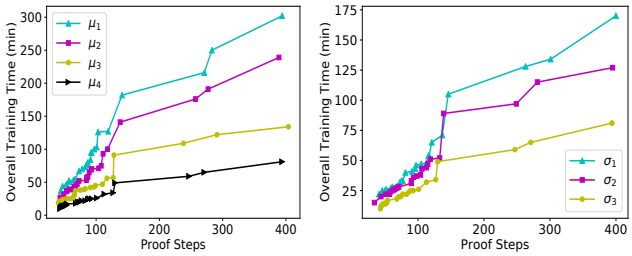
The experimental results for all the chosen protocols are shown in Table 1 and Figure 8. We use two metrics: **1)** the total time in searching; **2)** the quantity of nodes necessarily to be traversed (QN) when the searching succeeds, given the proof steps of verifying a security protocol. Here, for SmartVerif, the nodes includes which have been traversed during the Acquisition and Verification phases. Before comparison, an important observation is that it takes several seconds for a single step of new node traversing by using tamarin prover. It may take less time if using other tools, *e.g.*, the ProVerif-based tools. We also find that it takes more time when **a)** traversing a new node at the deeper level of tree, and **b)** initializing or reconfiguring the searching environment. For example, on verifying YubiKey protocol, the averaging time on traversing a node at level 10 and 100 is 0.2s and 0.4s, respectively, and the time on initialization is 1s. Therefore, the verification time of DFS and BFS tends to be affected by reason a) and b), respectively. Since the verification time may be affected by multiple factors, we also use QN as a complement metric in comparison.

A significant result is that the QN by DFS grows much faster than that of SmartVerif, when the proof steps increase starting from 65. Afterwards, the verification time of DFS reaches 48-hour limit when the proof steps are around 360. We further find that for most protocols with proof steps less than 60, DFS only needs to backtrack for less than 10 steps. For instance, for the TPM-toy protocol, DFS begins backtracking when it reaches the node at the depth 57, for the corresponding path is estimated incorrect. When succeeding in searching, the top 49 nodes in the incorrect path are the correct nodes representing supporting lemmata. Hence, when the depth for which DFS has to backtrack merely grows to more than 10, the performance of DFS starts to decrease drastically.

Therefore, SmartVerif greatly outperforms DFS when verifying complicated protocols. The QN of SmartVerif grows much more slowly when the proof steps increase. The phenomenon can be explained by our insight as illustrated in Section 7.3. Observe that the performance of BFS is even worse than the performance of DFS, though BFS runs in parallel. We omit the explanation due limitation of paper size.

Moreover, we implemented three naive algorithms as illustrated in Section 7.3, which use the built-in heuristics ('s', 'c' and 'p') of tamarin prover as the static strategy of selecting nodes respectively, DFS for tree traversing, and our module of correctness determination for back-traversing. As shown in Table 1, the comparative results are summarized as follows. **1)** For protocols like Yubikey, the naive algorithms still cannot succeed in automated verification. **2)** For protocols that cannot be verified by the original tamarin prover, SmartVerif achieves much better efficiency compared with the naive algorithms. **3)** For protocols that can be verified by the original tamarin prover, the naive algorithm only achieves similar performance with the original tamarin prover with the corresponding heuristics. **Discussion:** The results validates our analysis in Section 7.3. Here, an important observation is that for protocols of results 2), it is uncertain whether the naive algorithms with the built-in heuristics outperform the DFS without heuristics. An example is that Mobile-EMV protocol must be verified for at least 14 hours with the former algorithms, but it requires 7 hours for the latter algorithm. It can be inferred that the design of static strategies is non-trivial: an algorithm with a static strategy cannot be easily improved by leveraging other naive approaches, *e.g.*, DFS.

Note that we also measure the QN for all the chosen protocols using the heuristics ('s', 'c' and 'p') of tamarin prover. Since tamarin prover constructs proof paths based on static heuristics, for the protocols which can be verified by the heuristics of tamarin prover, the QN of tamarin with these heuristics is equal to the number of the proof steps of the correct proof path which is much less than the QN of SmartVerif. However, the static heuristics of tamarin prover cannot be



(a) Overall training time (b) Overall training time

Figure 9: Experimental results. (a): Overall training time using different number of GPUs. (b): Overall training time using different multithreading parameters.

used to verify all the protocols (*e.g.*, Yubikey and CANAuth) in our experiments.

In addition, we study the performance of training when using different number of GPUs. We try four sets of parameters. Here, μ_1 represents running SmartVerif without GPUs. μ_2 represents 0 graphic cards, which means that SmartVerif only use the integrated GPU in the CPU to compute in the training process. μ_3, μ_4 represents 2, 4 GTX 1080 Ti graphic cards respectively. As shown in Figure 9 (a), we can see an improvement to the overall training times when using more graphic cards in our experiment. For example, it only takes 59 minutes to verify the Yubikey protocol using four graphic cards. Using no GPUs, it takes 216 minutes to achieve a successful verification.

Furthermore, we evaluate the overall training times in verifying four protocols with different multithreading parameters. We try three sets of parameters. $\sigma_1, \sigma_2, \sigma_3$ represents 2, 4, 8 threads of Acquisition module executed in parallel respectively. As shown in Figure 9 (b), the running time is decreasing with the increasing quantity of threads executed in parallel for the parameters sets σ_1, σ_2 and σ_3. As shown in the above experimental results, SmartVerif achieve a solid performance on a high-performance server as well as a modest machine with less graphic cards.

8.2 Case Study

8.2.1 Yubikey Protocol

In the following, we briefly overview the Yubikey protocol SmartVerif verified. We provide some details in key steps of the verification. For the limitation of paper size, we do not detail all the formal models of the protocols and properties that we studied.

Kremer *et al.* [36] modeled and verified Yubikey protocol with unbounded sessions in tamarin prover. Specifically they define three security properties. All properties follow more or less directly from a stronger invariant. By default, tamarin prover cannot automatically prove this invariant, which is caused by a non-termination problem. To successfully verify

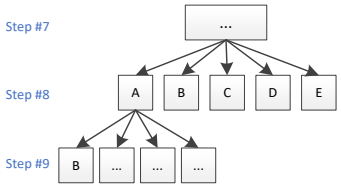


Figure 10: Part of the verification tree in Yubikey protocol

the protocol, tamarin needs additional human guidance, which is provided by experts in the interactive mode.

In the following, we analyze the choice made by tamarin prover, experts and SmartVerif. Specifically, in proof step #8, tamarin prover needs to select one rule, *i.e.*, lemma, from the rules as follows:

$$A : (\#vr.13 < \#t2.1) \parallel (\#vr.13 = \#t2.1) \parallel (\#vr.6 < \#vr.13))$$
$$B : State\_011111111111(lock11.1, n, n.1, nonce.1, npr.1, otc.1, secretid, tc2, tuple) \triangleright_0 \#t2)$$
$$C : Insert(<' Server', n>, <n.2, n.1, otc>)@\#t2.1)$$
$$D : !KU(n)@\#vk.2)$$
$$E : !KU(senc(<n.2, (otc+z), npr>, n.1))@\#vk.5)$$

Here, rule A is a restriction rule to the timepoints $\#vr.13, \#t2.1$ and $\#vr.6$. Rule B states an action $State\_011111111111$ must have been in the protocol execution in timepoint $\#t2$. Rule C states an action $Insert(<' Server', n>, <n.2, n.1, otc>)$ must have been in the protocol execution at timepoint $\#t2.1$. Rule D states the adversary has known the nonce n at timepoint $\#vk.2$. Rule E states the adversary has known the encrypted message $senc(<n.2, (otc+z), npr>, n.1)$ at timepoint $\#vk.5$.

Tamarin prover considers that rule A is a timepoint constraint rule, which is more likely to achieve a successful verification. It chooses the rule in the automated mode. Then, there are four rules to be chosen. By default, tamarin chooses the rule B. However, the rule leads to a loop in verification as follows:

$$State\_011111111111(lock11.1, n, n.1, nonce.1, npr.1, otc.1, secretid, tc2, tuple) \triangleright_0 \#t2$$
$$...$$
$$Insert(<' Server', n>, <n.2, n.1, otc>)@\#t2.1)$$
$$State\_011111111111(lock11.2, n, n.1, nonce.2, npr.2, otc.2, n.2, otc, tuple) \triangleright_0 \#t2.1$$
$$...$$
$$Insert(<' Server', n>, <n.2, n.1, otc.1>)@\#t2.2)$$
$$State\_011111111111(lock11.3, n, n.1, nonce.3, npr.3, otc.2, n.2, otc.1, tuple) \triangleright_0 \#t2.2$$

In this loop, tamarin prover keeps solving $Insert(<' Server', n>, <n.2, n.1, otc>)@\#t2.1)$ and $State\_011111111111(lock11.2, n, n.1, nonce.2, npr.2, otc.2, n.2, otc, tuple) \triangleright_0 \#t2.1$ rules alternately. It leads to non-termination in verification.

In interactive mode, experts make 23 manual rule selections to verify the protocol, and 11 of them are different from the one made by tamarin prover. Specifically, experts choose rule B as the supporting lemma at proof step #8, which leads to a successful verification.

In SmartVerif, we achieve a fully automated verification

Table 2: Q value of each rule in proof step #8.

	rule A	rule B	rule C	rule D	rule E
initial epoch	0	0	0	0	0
epoch 20	0.3	0.4	0.2	0.2	0.3
epoch 40	0.4	1.1	0.6	0.5	0.6
epoch 79	1.2	1.7	1.1	1.3	1.1
epoch 120	1.3	2.0	1.4	1.3	1.2

of Yubikey protocol without any user interaction. Figure 10 shows the corresponding part of the verification tree. The Q value of each rule in proof step #8 is shown in Table 2. In the initial epoch, the Q value of each rule is the same. In epoch 20, the network learns from its experience that candidate rules A, C, D, E may lead to non-termination cases with higher probability. Hence, the Q values of these rules have a slighter difference compared with Q value of rule B. Then, the difference between Q value of rule B and the Q value of other rules is getting larger in further epochs, which also validate our insight and the effectiveness of our designed strategy. In epoch 79, SmartVerif finds a correct proof path when choosing rule B. In further epochs, the difference among Q value of each rule is getting larger. Based on the Q values, SmartVerif finds the supporting lemma B automatically, such that the protocol can be verified without any user interaction.

8.2.2 CANAuth Protocol

We also investigate the case study presented by CANAuth protocol. Cheval *et al.* [17] encoded a model for the protocol.

In the following, we analyze the choice made by tamarin prover, human experts and SmartVerif. In proof step #10, tamarin prover needs to select one rule from the following rules:

$$A : solve((\#vr.29 < \#t2.1) | (\#vr.29 = \#t2.1))$$
$$B : solve(Insert(n.5, i)@\#t2.1)$$

Rule A states that timepoint $\#vr.29$ is earlier than or equals to $\#t2.1$. Rule B states action $Insert(n.5, i)$ is executed at timepoint $\#t2.1$.

Since the strategy of tamarin prover decides that the second rule is unlikely to result in a contradiction, it chooses rule A in the automated mode. However, the rule leads tamarin prover to a loop as follows:

$solve(State\_01111111111211111(lock7, n.5, cellB, i, msg.1, sk) \triangleright_0 \#t2.1)$
$solve(State\_01111111111211111(lock8, n.6, cellB, i, msg.2, sk) \triangleright_0 \#t2.2)$
$solve(State\_01111111111211111(lock9, n.7, cellB, i, msg.3, sk) \triangleright_0 \#t2.3)$
...

In interactive mode, experts make 4 manual rule selections to verify the protocol, and one of them is different from the selection made by tamarin prover. Specifically, experts choose rule B in proof step #10, which leads to success of the verification.

In SmartVerif, the result is similar to the previous case. Figure 11 shows the corresponding part of the verification tree. The Q value of each rule at proof step #10 as shown in Table 3. In the initial state, the Q value of each rule is the same. In epoch 10, the DQN discovers that candidate rule A may lead to incorrect paths. Hence, the Q value of rule A has a slighter difference compared with rule B. Then, in epoch 19, SmartVerif finds a correct proof path when choosing rule B. In epoch 100, the difference continues increasing.

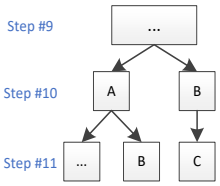


Figure 11: Part of the verification tree in CANAuth protocol.

Table 3: Q value of each rule in proof step #10.

	rule A	rule B
initial epoch	0	0
epoch 10	0.3	0.6
epoch 19	1.1	1.4
epoch 100	2.8	5.3

9 Limitation and Future Work

We currently train a standalone DQN for each studied protocol to keep a high level of generality. Another possible approach is to use pre-trained and optimized networks to verify protocols. However, it brings several challenges. Firstly, it is challenging to achieve a high level of accuracy on node selection in generating pre-trained network. Existing approaches generating pre-trained networks [51, 54] in a similar research field, *i.e.*, theorem proving, do not achieve a high level of accuracy on node selection. Compared with theorem proving, it is much challenging to generate pre-trained network with much higher accuracy, given much less samples of models of security protocols. Secondly, it is challenging to achieve high efficiency if using a generated pre-trained network. If using a pre-trained network, the verification time for some protocols may increase. For example, one could take the standard heuristic of tamarin prover as the basic strategy in our DQN to verify security protocols. However, in this case, the DQN does not optimize itself in an efficient way when verifying complicated protocols like Yubikey protocol. Therefore, we train a standalone DQN for each studied protocol. Similarly we currently retrain the DQN when verifying a new security property of a protocol. Therefore, it requires to retrain the DQN after modifying the protocol or the property specification during practical usage. We will try to optimize the network design and use other learning techniques in future work.

Our work opens several directions for future work. **1)** *Hybrid strategy*. Since the initial strategy in SmartVerif is purely random, the strategy may be optimized with less epochs if it is implemented with some static strategy. However, the problem is still challenging that there is a potential risk that the epochs may become larger for some special protocols that the static

strategy does not support. **2)** *Scalability*. It is possible that our dynamic strategy can be used to cope with more complicated problems, such as automated formal verification of software or systems [18, 46] that are based on first-order logics [50] or higher-order logics [8]. They are quite similar that they can be translated into a path searching problem. We will also explore and verify more complicated security protocols using SmartVerif. **3)** *Efficiency*. Currently, we train a standalone DQN for each studied protocol to keep a high level of generality. Designing a universal network which can verify all the protocols may increase the efficiency and improve the performance of SmartVerif. Therefore, we will try to optimize the network design and use other AI techniques in future work.

10 Conclusion

In this paper we have studied automated verification of security protocols. We propose a general and dynamic strategy to verify protocols. Moreover, we implement our strategy in SmartVerif, by introducing a reinforcement learning algorithm. As demonstrated through experiment results, SmartVerif automatically verifies security protocols that is beyond the limit of existing approaches. The case studies also validate the efficiency of our dynamic strategy.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ralf Sasse for providing us valuable feedback for improving our paper. The research is supported by the National Key R&D Program of China 2018YFB0803400, 2018YFB2100300, National Natural Science Foundation of China under Grant No.61972369, No.61572453, No.61520106007, No.61572454, and the Fundamental Research Funds for the Central Universities, No. WK2150110009. Wenchao Huang is the corresponding author of the paper.

References

[1] BNB ERC20 Whitepaper. https://www.binance.com/resources/ico/Binance_WhitePaper_en.pdf, 2019.

[2] (Not So) Smart Contracts Official Repository. https://github.com/crytic/not-so-smart-contracts, 2019.

[3] SmartVerif. https://www.dropbox.com/sh/u7stf9cqwujytp1/AAAN3V6CKTAf8cPTSVSLHUbZa?dl=0, 2019.

[4] Tamarin Prover Official Repository. https://github.com/tamarin-prover/tamarin-prover, 2019.

[5] The TLS 1.3 Model. https://github.com/tls13tamarin/TLS13Tamarin/blob/master/src/rev21/proofs/session_key_agreement.spthy, 2019.

[6] Myrto Arapinis, Eike Ritter, and Mark Dermot Ryan. Statverif: Verification of stateful processes. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 33–47, 2011.

[7] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 281–285, 2005.

[8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 186–197, 2012.

[9] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1383–1396, 2018.

[10] David A. Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1144–1155, 2015.

[11] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P'17)*, pages 483–503, San Jose, CA, May 2017. IEEE. Distinguished paper award.

[12] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 82–96, 2001.

[13] Alessandro Bruni, Sebastian Mödersheim, Flemming Nielson, and Hanne Riis Nielson. Set-pi: Set membership p-calculus. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 185–198, 2015.

[14] Alessandro Bruni, Michal Sojka, Flemming Nielson, and Hanne Riis Nielson. Formal security analysis of the macan protocol. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, pages 241–255, 2014.

[15] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles, SOSP 1989, The Wigwam, Litchfield Park, Arizona, USA, December 3-6, 1989*, pages 1–13, 1989.

[16] David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of CRYPTO '82, Santa Barbara, California, USA, August 23-25, 1982.*, pages 199–203, 1982.

[17] Vincent Cheval, Véronique Cortier, and Mathieu Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF'18)*, 2018.

[18] David Cock, Qian Ge, Toby C. Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on sel4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 570–581, 2014.

[19] Véronique Cortier, Alicia Filipiak, Jan Florent, Said Gharout, and Jacques Traoré. Designing and proving an emv-compliant payment protocol for mobile devices. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 467–480, 2017.

[20] Véronique Cortier, David Galindo, and Mathieu Turuani. A formal analysis of the neuchatel e-voting protocol. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 430–442, 2018.

[21] Cas Cremers, Martin Dehnel-Wild, and Kevin Milner. Secure authentication in the grid: A formal analysis of DNP3: sav5. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, pages 389–407, 2017.

[22] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1773–1788, 2017.

[23] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-rtt, resumption and delayed authentication. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 470–485, 2016.

[24] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[25] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. Formal analysis of protocols based on TPM state registers. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 66–80, 2011.

[26] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal security analysis of pkcs#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, 2010.

[27] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Information Theory*, 29(2):198–207, 1983.

[28] Santiago Escobar, Catherine A. Meadows, and José Meseguer. Maude-npa: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, pages 1–50, 2007.

[29] Martin Feldhofer, Sandra Dominikus, and Johannes Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pages 357–370, 2004.

[30] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1204–1215, 2016.

[31] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *Advances in Cryptology - AUSCRYPT '92, Workshop*

on the Theory and Application of Cryptographic Techniques, Gold Coast, Queensland, Australia, December 13-16, 1992, Proceedings, pages 244–251, 1992.

[32] Juan A. Garay, Markus Jakobsson, and Philip D. MacKenzie. Abuse-free optimistic contract signing. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 449–466, 1999.

[33] Joshua D. Guttman. State and progress in strand spaces: Proving fair exchange. *J. Autom. Reasoning*, 48(2):159–195, 2012.

[34] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath - deep sequence models for premise selection. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2235–2243, 2016.

[35] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*, pages 435–450, Paris, France, April 2017. IEEE.

[36] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 163–178, 2014.

[37] Robert Künnemann and Graham Steel. Yubisecure? formal security analysis results for the yubikey and yubihsm. In *Security and Trust Management - 8th International Workshop, STM 2012, Pisa, Italy, September 13-14, 2012, Revised Selected Papers*, pages 257–272, 2012.

[38] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[39] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, pages 147–166, 1996.

[40] Catherine A. Meadows. Language generation and verification in the NRL protocol analyzer. In *Ninth IEEE Computer Security Foundations Workshop, March 10 - 12, 1996, Dromquinna Manor, Kenmare, County Kerry, Ireland*, pages 48–61, 1996.

[41] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 696–701, 2013.

[42] Jonathan K. Millen, Sidney C. Clark, and Sheryl B. Freedman. The interrogator: Protocol security analysis. *IEEE Trans. Software Eng.*, 13(2):274–288, 1987.

[43] John C Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using mur/spl phi. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 141–151. IEEE, 1997.

[44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[45] Sebastian Mödersheim. Abstraction by set-membership: verifying security protocols and web services with databases. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 351–360, 2010.

[46] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 415–429, 2013.

[47] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[48] OASIS Standard. Pkcs #11 cryptographic token interface base specification version 2.40, 2015.

[49] David J. Otway and Owen Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[50] Raja Oueslati and Olfa Mosbahi. Distributed reconfigurable b approach for the specification and verification of b-based distributed reconfigurable control systems. *Advances in Mechanical Engineering*, 9(11):1687814017730731, 2017.

[51] Bartosz Piotrowski and Josef Urban. Atpboost: Learning premise selection in binary setting with atp feedback. *Lecture Notes in Computer Science*, page 566–574, 2018.

[52] John D. Ramsdell, Daniel J. Dougherty, Joshua D. Guttman, and Paul D. Rowe. A hybrid analysis for security protocols with state. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, pages 272–287, 2014.

[53] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 78–94, 2012.

[54] Taro Sekiyama and Kohei Suenaga. Automated proof synthesis for propositional logic with deep neural networks. *CoRR*, abs/1805.11799, 2018.

[55] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, pages 926–934, 2013.

[56] Anthony Van Herrewege, Dave Singelee, and Ingrid Verbauwhede. Canauth-a simple, backward compatible broadcast authentication protocol for can bus. In *ECRYPT Workshop on Lightweight Cryptography*, volume 2011, 2011.

[57] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1313–1328, 2017.

[58] Guilin Wang. Generic non-repudiation protocols supporting transparent off-line TTP. *Journal of Computer Security*, 14(5):441–467, 2006.

[59] Jorden Whitefield, Liqun Chen, Frank Kargl, Andrew Paverd, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. Formal analysis of V2X revocation protocols. In *Security and Trust Management - 13th International Workshop, STM 2017, Oslo, Norway, September 14-15, 2017, Proceedings*, pages 147–163, 2017.

[60] Yubico AB. The yubikey manual - usage, configuration and introduction of basic concepts (version 3.4), 2015.

A Proof of Our Insight

We prove our insight of the paper that the node representing a supporting lemma is on the incorrect path with lower probability, when a random strategy is given. To illustrate our insight more comprehensively, we translate the complicated verification process into a path searching problem. Here, the verification can be simply regarded as the process of path searching in a tree: each node represents a proof state which includes a lemma as a candidate used to prove the lemma in its father. The supporting lemma is a special lemma necessarily used for proving the specified security property.

Formally, we use $[n_{t_1}, n_{t_2}, ..., n_{t_{k_t}}]$ to denote a proof path t, where n_{t_i} is the ith node in the path and k_t is the number of nodes in the path. Therefore, the lemmata in $\{n_{t_i}\}$ are the candidate lemmata. For a node n_{t_i}, we use x_{t_i} and y_{t_i} to denote its degree and the number of its children which represents supporting lemmata respectively. Suppose there are at least one child of n_{t_i} that does not represent supporting lemma, *i.e.*, $x_{t_i} > y_{t_i}$. The random strategy here means that whenever choosing a child for searching, the probability of choosing is uniform. In other words, the probability of choosing any child of n_{t_i} is $\frac{1}{x_{t_i}}$. Hence, if the random strategy is applied in choosing a child of n_{t_i}, the probability of choosing the node representing supporting lemma is $\frac{y_{t_i}}{x_{t_i}}$. Suppose there are R correct and complete proof paths in a given tree.

Theorem 1. *Given the above assumptions, if n_{t_i} has been chosen, the node representing a supporting lemma, who is the child of n_{t_i}, is on an incorrect path with the probability less than $\frac{y_{t_i}}{x_{t_i}}$, when the random strategy is given.*

Proof. For the rth correct and complete path, define α_r as follows:

$$\alpha_r = \begin{cases} \prod_{j=i}^{k_r-1} \frac{1}{x_{r_j}} & \text{if } \forall j \in [1,i].n_{r_j} = n_{t_j} \\ 0 & \text{otherwise} \end{cases}$$

Denote p_1 as the probability that a selected path is incorrect if n_{t_i} has been chosen.

$$p_1 = 1 - \sum_{r=1}^{R} \alpha_r$$

Denote $m_1, m_2, \ldots, m_y$ as the nodes representing supporting lemmata among the children of n_{t_i}. For the rth correct and complete path, define $\beta_{r,s}$ as follows:

$$\beta_{r,s} = \begin{cases} \prod_{j=i+1}^{k_r-1} \frac{1}{x_{r_j}} & \text{if } n_{r_{i+1}} = m_s \wedge \forall j \in [1,i].n_{r_j} = n_{t_j} \\ 0 & \text{otherwise} \end{cases}$$

It can be inferred that

$$\alpha_r = \sum_{j=1}^{y_{t_i}} \frac{1}{x_{t_i}} \beta_{r,j}$$

Denote p_2 as the probability that a selected path is incorrect and the child of n_{t_i} representing supporting lemma is on the path if n_{t_i} has been chosen.

$$p_2 = \sum_{j=1}^{y_{t_i}} \frac{1}{x_{t_i}} (1 - \sum_{r=1}^{R} \beta_{r,j})$$

Therefore, denote p as the probability that a child of n_{t_i} representing supporting lemma is on an incorrect path if n_{t_i} has been chosen.

$$p = \frac{p_2}{p_1} = \frac{\sum_{j=1}^{y_{t_i}} \frac{1}{x_{t_i}} (1 - \sum_{r=1}^{R} \beta_{r,j})}{1 - \sum_{r=1}^{R} \sum_{j=1}^{y_{t_i}} \frac{1}{x_{t_i}} \beta_{r,j}} = \frac{y_{t_i} - \sum_{r=1}^{R} \sum_{j=1}^{y_{t_i}} \beta_{r,j}}{x_{t_i} - \sum_{r=1}^{R} \sum_{j=1}^{y_{t_i}} \beta_{r,j}} < \frac{y_{t_i}}{x_{t_i}}$$

□

As a result, given the random strategy, the probability that n_i is on an incorrect path is less than the probability that n_i is on a given path. In other words, if an incorrect path is found, the probability that n_i is on the path, which equals $\prod_{j=1}^{i-1} \frac{1}{x_j}$ on a given path, decreases. On the other hand, the DQN requires a reward for guiding the optimization, where a reward corresponds to a determined occurrence of an event, *e.g.*, a dead-or-alive signal upon an action in a game [44]. However, there is no such determined event in verifications. Instead, in SmartVerif, we leverage probability of occurrence that the node representing a supporting lemma is on incorrect paths for constructing the reward according to Theorem 1. This insight enables us to leverage the detected incorrect paths to guide the path selection, which is implemented by using the DQN.

B Technical Details - Deep Q Network

Algorithm 2 demonstrates the technical details of our implementation of DQN. The DQN runs iteratively with multiple epochs. In each epoch, recalling that we adopt a multi-threading approach for increasing the efficiency, the DQN launches σ threads in which the paths are selected according to the policy (line 5). If a path is estimated correct and complete, SmartVerif terminates with the proof path (line 6). If all the selected paths are estimated incorrect, the policy is optimized (line 7).

In path selection, we use two strategies in the policy (line 12): 1) an exploration strategy to choose random actions, which is to explore the values of unchosen actions; 2) a greedy strategy to choose a which may have the largest Q value currently. Here, $Q(s_t, a; \theta_e)$ is a pre-defined function [44] that outputs comparable value, given the node s_t and its ath child. The Q function also takes θ_e as input, where θ_e is the set of the DQN's parameters at epoch e, and θ_e is updated into θ_{e+1} in policy optimization. Combining the two strategies, we use a ε-greedy strategy to select actions. Here, ε is a probability value for selecting random actions. We change the value of ε to get different exploration ratios. Note that we choose random actions in the exploration strategy. Another possible approach is to take standard heuristic of tamarin prover as the basic strategy. However, for example, when verifying Yubikey protocol, the standard heuristic does not rank the supporting lemma at the first place in several proof steps. In this case, the DQN does not optimize itself in an efficient way and the efficiency is worse than SmartVerif.

Algorithm 2 Implementation of DQN

```
1:  Initialize a replay memory D to capacity N
2:  Initialize an action-value function Q
3:  success = 0
4:  for e = 1 to EPOCH do
5:      Call σ threads that execute path_selection
6:      if success = 1 then Program ends
7:      Execute policy_optimization
8:
9:  function path_selection:
10: Initialize a proof state s_1
11: for t = 1 to ROUND do
12:     With probability ε select a random action a_t
        otherwise select a_t = max_a Q(s_t, a; θ_e)
13:     Generate next state s_{t+1} according to a_t
14:     Store a transition (s_t, a_t, ω, s_{t+1}) in D
15:     if the path is estimated incorrect then break
16:     if the path is estimated correct and complete then
17:         success = 1
18:         return
19:
20: function policy_optimization:
21: Sample n random transitions (s_j, a_j, r_j, s_{j+1}) from D
22: Set y_j = r_j + γ max_{a'} Q(s_{j+1}, a'; θ_e)
23: Perform a gradient descent step on (y_j − Q(s_j, a_j; θ_{e+1}))^2
```

To apply our insight, we set the reward to the same negative number for all the edges on each estimated incorrect proof path. Specifically, in line 14, a transition, *i.e.*, tuple $(s_t, a_t, \omega, s_{t+1})$, is generated and added to D, where w is the negative reward for the action a_t at the state s_t. D is a replay memory [38] with capacity N, *i.e.*, in practice, our network only stores the last N tuples in the replay memory.

In policy optimization, θ_e in Q function is updated as mentioned (line 20). Here, n tuples are randomly selected from D. For each selected tuple (s_j, a_j, r_j, s_{j+1}), we compute y_i according to θ_e. Then θ_{e+1} is estimated by using the loss function $(y_i - Q(s, a, \theta_{e+1}))^2$.