FORMAL SECURITY ANALYSIS: SECRECY, AUTHENTICATION AND ATTESTATION

LI LI
(B.Sc., Huazhong University of Science and Technology, 2011)

A THESIS SUBMITTED

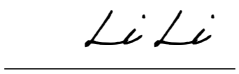
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2015

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



Li Li

30 November 2015

Acknowledgements

First and foremost, I am deeply indebted to my supervisor, Dr. Dong Jin Song, who guide me with many precious advice and encourage me with lots of humor throughout the course of my PhD study. He has given me immense support in various ways from academic research to personal life. I could not have wished for a better supervisor. I am deeply grateful to Dr. Sun Jun, Dr. Liu Yang and Dr. Pang Jun, who act like both friends and co-supervisors in my graduate years. Their supervision and contribution is the backbone of my graduate research. I really enjoyed our research and I am looking for more in the future.

Furthermore, I would like to thank my thesis advisory committees, Professor Chang Ee Chien, Professor Liang Zhenkai and Professor Steve Schneider, for their invaluable comments and suggestions on my research works. I would like to thank the chair of my thesis defense, Professor Hugh Anderson, for his active participation and constructive feedback.

I would like to thank all the current and former members of the PAT group. Especially, I would like to thank Dr. Song Songzheng, Dr. Shi Ling, Dr. Liu Yan, Dr. Tan Tian-huat, Dr. Truong Khanh Nguyen, Dr. Gui Lin, Dr. Liu Shuang, Dr. Bai Guangdong, Dr. Chen Manman for their help on my research works. I have special thanks to Dr. Sun Meng, Dr. Li Xiaohong, Dr. Dong Naipeng, Mr. Hu Hong, Mr. Xie Xiaofei, etc. for our research collaborations. Thank you for your support and friendships through my PhD study.

I thank sincerely and deeply my parents, Li Shu Qiang and Song Li Ping, who have

taken great care of me with unconditional love. I would like to thank all my friends. They keep me balanced and well-earthed despite the rather abstract and sometimes demanding nature of my PhD.

Abstract

In cyber security systems, various security protocols have been developed to provide trustworthy communications. However, designing security protocol is challenging and error-prone, which is well illustrated by many security protocols attacks. Hence, it is necessary to provide a verification framework where the security protocols can be formally checked. In this thesis, we first analyze a vehicle charging protocol to show the strengths and weaknesses of existing methods. Then, we propose a verification framework, where the security protocols can be intuitively specified and efficiently verified. Comparing with the existing methods, our verification method requires no abstraction during the verification and works for an unbounded number of protocol sessions. Security protocols in real-world use not only cryptography but also physical properties. Hence, we develop a generic analysis method to the protocols that consider physical properties. We analyze a family of software-based attestation protocols using this method and find several security weaknesses.

Contents

Summary

Trustworthy communications are needed in the cyber systems. In order to provide the security, various security protocols are developed. However, designing security protocol is challenging and error-prone, which is well illustrated by many attacks found in the security protocols. Hence, it is necessary to provide a verification framework where the security protocols can be formally checked.

In our first work, we analyze a complex vehicle charging protocol to show the strengths and weaknesses of existing verification methods. Many interesting properties are analyzed such as secrecy, authentication and privacy. The analysis shows that manual modeling abstractions are generally needed to ensure the termination of the verification, making the protocol specification less intuitive and more laborious. Additionally, false alarms can be introduced when verification considers specific domain knowledge.

Then, in our second work, we propose a security protocol verification framework, where the security protocols can be intuitively specified and efficiently verified. Comparing with the existing methods, our verification method requires no abstraction during the verification and works for an unbounded number of protocol sessions. Our research shows that this framework can be applied to specific domains, e.g., the timed domain, easily. The correctness of our verification algorithm has been formally proved. We implement our method into a tool and use it to verify many timed and untimed security protocols successfully.

Security protocols in real-world use not only cryptography primitives but also physical

properties. Hence, we investigate a family of software-based attestation protocols that are in this category so as to provide an analysis approach for them. These attestation protocols, comparing with traditional hardware-based (e.g., TPM-based) attestation protocols, do not require any hardware support in the attestation. Instead, their security are provided by the computation limitation of the target devices. We analyze these protocols in three stages. First, we propose a generic specification for them that captures most existing software-based attestation protocols. Then, we formalize various security criteria that should be satisfied by the generic scheme. Finally, we apply the security criteria back to the existing software-based attestation protocols. Using this approach, we find several security flaws successfully.

List of Tables

List of Figures

Chapter 1

Introduction

Security protocols usually are short programs that use cryptographic primitives to provide secure communications in insecure networks, such as the Internet and the networks used in the cyber-physical systems. They are widely used in our daily lives in, e.g., bank transactions, mobile phones, wifi networks, e-votings. However, these security protocols (and their manual proofs) are error-prone, which is well illustrated by the famous Needham-Schroeder public-key protocol [93] where a security attack is found by Gavin Lowe 18 years after its publication [83]. Many new attacks are still being discovered continuously to the manually proved protocols [4, 6, 35, 105, 100, 61]. As security protocols are designed to provide security, their design flaws usually have serious consequences, which can lead to the loss of money or even human lives. Even though correctness proofs for security protocols can be given manually [46], manual proving their correctness is extremely hard when we consider (1) an active network attacker who controls the whole network, as well as (2) infinitely many protocol sessions running in parallel at the same time. Hence, an automatic verification framework for the security protocols that can generate machine proofs automatically would be extremely useful.

However, developing a fully automatic approach to verify security protocols is chal-

lenging. First, we need to provide a consistent specification method for describing the security protocols and their security properties. Second, we need to develop a verification algorithm that can either prove the correctness of the protocols or generate attack instances. In order to prove the correctness of security protocols, we need to consider an unbounded number of protocol sessions that are running at the same time during the verification. Since verifying the security protocols in this scenario is undecidable in general [41] (the termination of the verification cannot be ensured), we need to develop an algorithm that can terminate on most of the practically used security protocols. Third, our framework should be able to be applied to specific domains. For instance, as many security protocols consider time and they might be vulnerable under the timing attacks, our framework should be capable of verifying timed security protocols.

1.1 Literature Review

Many symbolic verification logics are proposed to check security protocols using, e.g., Horn logic, strand space logic and multiset rewriting logic.

Horn Logic. Using Horn logic to verify security protocol was first proposed by Weidenbach [119]. Later in [29], Blanchet extended Weidenbach's method with abstraction of nonces, so that Horn logic can be used to specify and verify security protocols practically. Their approach generally works as follows. First, they represent adversary capabilities with Horn logic rules. For instance, the decryption capability of the adversary can be represented by the rule $enc_s(m,k), k \rightarrow m$, where enc_s is a symmetric encryption function, m is a message and k is a key. It means that if the adversary knows the encryption $enc_s(m,k)$ and the key k then he can get the plaintext m using the decryption function. In this method, the security protocols can be formalized by the adversary's capabilities so as to facilitate efficient verification against various attacks. Then, a verification algorithm, similar to the forward searching, is proposed to check the secrecy property of messages.

Comparing with the traditional forward searching where a rule's conclusion is reachable when its premises are satisfied, this new searching algorithm treats the rule's conclusion as reachable when all of the premises in the rule are variables. For example, suppose we have two rules such that $m, k \rightarrow enc_s(m, k)$ and $enc_s(A, k) \rightarrow S$, where m is a message, k is a key, A and S are two constants. The verification algorithm takes $enc_s(m, k)$ in the first rule as reachable because its premises are variables. Then, the algorithm uses the conclusion from the first rule to fulfill the premise of the second one, generating a new rule $A, k \rightarrow S$. This new rule means that the constant S is reachable whenever the constant A is reachable. The variable k can be ignored because the adversary can use any message as k (it is an unrelated variable). In addition, in order to help the verification to terminate, the algorithm proposed in [29] over-approximates the protocol execution by merging the nonces when the protocol sessions have the same input sequence. Even though finding attacks and proving correctness for security protocols are undecidable in general, the verification method proposed in [29] can be used to analyze real-world security protocols and terminate quickly.

In [30], Blanchet proposed to extend the secrecy checking to verify two types of authentication properties, i.e., non-injective agreement and injective agreement. In order to clearly specify the authentication properties, Blanchet uses an *init* event to represent the initialization of a protocol session and an *accept* event to stand for the acceptance of a protocol session. Then, the non-injective agreement means that for every *accept* event emitted, there exists a corresponding *init* event engaged before. The injective agreement additionally requires that every *init* event can correspond to at most one *accept* event. By following the secrecy checking algorithm, they can verify the non-injective agreement using a reachability checking with two phases. In the first phase, they need to ensure that the *accept* event is reachable. Then, in the second phase, they need to ensure that that *accept* event can only be reached after the corresponding *init* event is engaged. Furthermore, the injective agreement is satisfied if the following two conditions are satisfied. First, for

every *accept* event engaged in the protocol, the corresponding *init* event must be engaged before. Second, for every *init* event engaged in the protocol, at most one corresponding *accept* event can be engaged later.

Strong secrecy [31] is an observational equivalence property, that can be verified using Horn logic as well. It ensures that the adversary cannot observe any difference between two protocol instantiations which differ in some secret values. In [31], they introduced a predicate $testunif$ to the Horn logic. $testunif(p, p')$ is true if and only if $p \neq p'$ and there exists a substitution σ that $\sigma p = \sigma p'$ by substituting the secret value and the adversary can get the terms for the substitution. Obviously, when $testunif(p, p')$ is true, the adversary can send the terms to the protocol to distinguish the values used in the protocol sessions. In addition to the strong secrecy, Blanchet et al. proposed a way to check selected equivalence [32], which ensures observational equivalence when the changed secret value is selected from a finite set. Selected equivalence should be preserved by protocols such as the e-voting, where a voter's vote should be unobservable by the adversary and the possible vote (candidate) are selected from a finite set of values. In [31] and [32], the observational equivalence can only be checked against two protocol instantiations of the same structure. Cheval et al. [43] proposed to check observational equivalence for protocols with different structures by imposing a *fail* case in verification. So given two structurally different protocols, they are observational equivalent if they both fail or they satisfy the selected equivalence.

Strand Space. In [118], Thayer et al. proposed *strand space* to check security protocol. A strand represents a protocol interaction trace, consisting of a sequence of events such as sending, receiving, encrypting and decrypting messages. Given a limited number of strands, there exist a limited number of ways for them to interact with each other. The state space of combining the strands is thus called strand space. Hence, by exhausting the states in the strand space, either the protocol is proved as secure or an attack is found.

Notice that the proving process is not automated in [118]. Later, Song [112] proposed a method to automate the verification process using strand processes. They implement their method into a tool named Athena. Model checking technique is adopted to formalize the protocol execution and pruning theorems are employed to prunes the search space which increase its verification efficiency. The weakness of this work is that they could only give verification to security protocol with a bounded number of sessions.

Cremers followed Song's work and developed a tool named Scyther [48]. Scyther can verify security protocols with an unbounded number of sessions if a finite number of patterns, the ways that the strands can compose together, can be found during the verification. Even though Scyther does not give unbounded verification to all security protocols, it has the advantage in formalizing various adversary models that could be stronger than Delov-Yao model [56]. In the Delov-Yao model, the adversary has full control over the network, but he cannot compromise the keys and messages used in the protocol. When forward secrecy, weak forward secrecy, etc. are required, we need to verify the protocols against the adversaries who can compromise keys and messages. By using the strand space, these adversaries can be modeled in a straight forward manner. In [49, 24, 118, 50, 25, 26, 51], Cremers et al. propose several extensions to Scyther that consider different attack models in the verification.

Multiset Rewriting. Multiset rewriting was first proposed by Cervesato et al. [41] to specify and verify security protocols using multiset rewriting rules. Multiset rewriting considers the protocol execution as rewriting terms in the knowledge base. However, multiple rewriting rules can be applied at the same time. Schmidt et al. [104] uses multiset rewriting to specify protocols and adversary capabilities. In addition, a guarded fragment of first-order logic is adopted to specify security properties. Furthermore, equational theories are employed to model the algebraic properties of cryptographic operators. Schmidt et al. developed a tool named Tamarin [89] based on this method.

Tamarin has the specification flexibility of Scyther where the sessions can be explicitly specified in the model. Comparing with Scyther where different adversary model is built-in and fixed, the adversary model in Tamarin can be specified by the users directly using multiset writing rules. Even though Tamarin is strong in protocol specification capability, modeling protocols with multiset rewriting rules directly is hard and error-prone for the end-users. In general, multiset rewriting approach is a combination of the Horn logic approach and the strand space approach. It divides the trace of the protocol execution into small fragments so that protocols can be verified more efficiently than using strand space. Additionally, it preserved the trace information so that interesting properties, e.g, forward secrecy, can still be easily specified and verified. In [73], Kremer et al. proposed to translate stateful applied π-calculus to multiset rewriting rules and use Tamarin [89] as its backend to verify stateful protocols.

1.2 The Objectives and Contributions

The research gaps for symbolic verification of security protocols are listed as follows.

- Several symbolic verification tools [29, 48, 89] based on different theoretical foundations, e.g., Horn logic, strand space logic and multiset rewriting logic, are developed. They have been used to analyze many security protocols successfully. However, given a certain security protocol and a specific security property, it is unclear which tool is better to use for the verification. It is also interesting to investigate whether they have weaknesses in verifying certain types of protocols.

- Since the verification of security protocols is undecidable, many tools either need abstraction in the verification which gives false alarms, or can only handle a bounded number of protocol sessions. In addition, when verifying security protocols in specific domains, some abstraction can lead to meaningless verification results. For instance, in ProVerif [29], the nonces with the same name are merged even if they

are generated in different sessions, when they have the same session interaction history. So, given an expired session key (a nonce) generated in the previous session, it may can be used in a new session because the session key used in the new session remains the same. Hence, when an expired session key (a nonce) is merged with an unexpired session key generated in a new session, the expired session key can still be used in the new session since they share the same value. When a verification tool adopted this abstraction, a false alarm will be triggered saying that an expired session key is accepted in the protocol. However, this attack cannot be conducted in the real protocol because the nonces generated in different sessions should be different. So we need to find a new verification method without the abstraction when time is involved in the protocol verification.

- Given a symbolic verification framework, verifying security protocols used in practice is still challenging, as their designs consider the execution environment where physical properties and hardware features play important roles.

In this thesis, we thus wants to study above research gaps and develop techniques to bridge the gaps. The contributions of this thesis are listed as follows.

- In order to show the strengths and weaknesses of different verification approaches, we analyze a motivating security protocol that considers many security properties such as the secrecy, authentication and privacy. The analysis reveals many problems in existing tools. First, manual modeling abstractions are generally needed to ensure the termination of the verification, making the protocol specification effort less intuitive and more laborious. Second, false positives can be introduced when verification considers specific domain knowledge. Third, the verification for an unbounded number of sessions cannot always terminate for medium or even small sized protocols with existing tools.

- Based on the case study, we propose a security verification framework that can be flexibly extended to specific domains, for instance, the timed domain. The security protocols can be specified by either *timed logic rules* or *timed applied π-calculus* in our framework. In order to verify the timed security protocols, we ensure that no abstraction is made during the verification. The correctness of security protocols are formally proved. We implement our method into a tool and use it to verify many timed and untimed security protocols successfully.

- Security protocols in real-world tend to be more complex, which use their execution environment (i.e., hardware performance) to provide security. In order to develop an analysis approach that works for protocols in this case, we investigate a family of software-based attestation protocols that fall into this category. These attestation protocols, comparing with traditional hardware-based (e.g., TPM-based) attestation protocols, do not require any hardware support in the attestation. Instead, their attestation are based on the computation limitation of the target devices. We analyze these protocols in three stages. First, we propose a generic specification for software-based attestation protocols that captures most existing software-based attestation protocols. Then, we formalize various security criteria that should be satisfied by the generic scheme. Finally, we apply the security criteria back to the existing software-based attestation protocols. Using this approach, we find several security flaws successfully.

1.3 Publications

The work in Chapter 2 was published in the proceeding of the 19th international conference on engineering of complex computer systems [76]. The work in Chapter 3 was published in the proceeding of the 16th international conference on formal engineering methods [77]. The work in Chapter 4 was originally published in the proceeding of the

20th international symposium on formal methods [78], and its journal version is about to be submitted to IEEE transactions on software engineering. Additionally, the work in Chapter 5 was published in the proceeding of the 16th international conference on formal engineering methods [75].

Overview. The goal of this thesis is to verify security protocols where both of cryptography primitives and physical properties are used. I first present a case study on an electric vehicle charging protocol (EVCP) to evaluate the existing tools. EVCP involves many complex security properties, including secrecy, authentication and location privacy. Hence, I used it as an example for verifying cryptography and physical properties together. Then, I propose a symbolic verification method for timed protocols that are more related to cryptography in Chapter 3 and Chapter 4. In Chapter 5, I propose a generic analysis method for software-based attestation protocols that are more related to physical properties. Then, future works are discussed in Chapter 6.

Chapter 2

Evaluation of Existing Tools: Symbolic Analysis of an Electric Vehicle Charging Protocol

In this chapter, we describe our analysis of a recently proposed electric vehicle charing protocol [80]. The protocol builds on complex cryptographic primitives such as commitment [96], zero-knowledge proofs, BBS+ signature [16] and etc. Moreover, interesting properties such as secrecy, authentication, anonymity, and location privacy are claimed in this protocol. It thus presents a challenge for formal verification, as no single existing tool for security protocol analysis can verify all the required properties. In our analysis, we employ and combine the strength of two state-of-the-art symbolic verifiers, Tamarin and ProVerif, to check all important properties of the protocol.

2.1 Introduction

Electric vehicles are promising and futuristic automobiles which use electric batteries for clean energy. They dominate conventional vehicles from several aspects such as air pollution reductions, less power emissions and lower oil dependencies. In order to support the wide adoption of electric vehicles, Vehicle- to-Grid (V2G) is proposed. In V2G, it

is possible and highly recommended to do charging when the demand is low, especially after midnight, and to send the electricity back (recharging) to the system during the peak time. Despite these advantages, one of the concerns is the potential privacy leakage along with the charging route. Since energy storage devices cannot meet the requirement for long-term driving, electric vehicles need to visit charging stations frequently for energy supplying. As a consequence, the location privacy disclosed along with the charging and recharging behaviors has drawn particular attentions.

Due to this specific application requirement, a privacy-preserving electric vehicle charging protocol (ECP) has been recently designed by Liu et al. [80]. In this protocol, various complex cryptographic primitives are used and many security properties are claimed. Firstly, the user could make *commitment* to some data and expose the key later to open the commitment. It requires that the secrecy properties of the keys are related to the order of events happened in the protocol. Specifically, the commitment scheme requires that the private opening keys for the commitments should be unknown to the supplier until the user exposes them explicitly in the protocol. Secondly, this protocol supports the *two-way transmission*, which means the users are allowed to charge their vehicle at the stations, as well as recharge the electricity back to the stations with their balance refunded. This is achieved using multiple generators in the commitment scheme which is *homomorphic*, such that operations could be made on commitment to change the balance without knowing the explicit value. In addition, the supplier is potentially *dishonest* in this protocol. *Injective agreement* between the user and the supplier should always be guaranteed, which ensures that the supplier could only charge the users just as he should. Fourthly, the protocol is *stateful* in which manipulations over global mutable state are required. In this protocol, each user gets an account state after the registration. He needs to use the information stored in the state to communicate with the supplier in the later sessions and update them after each successful transaction.[1] Lastly, privacy properties such as

[1] As a result, some security protocol checkers could not terminate or even specify this protocol because of

anonymity and location privacy should be preserved by the protocol against the supplier. Anonymity makes sure that the supplier could not get any partial information about the users when they charge and recharge at the stations. Meanwhile, location privacy ensures that the supplier could not identify the station where users perform charing or recharging. In this protocol, privacy properties are achieved by using the zero-knowledge proofs and BBS+ signature [16] with commitment scheme.

Even though most of security protocol are designed carefully and manual proofs are given along with their publication, the protocols as well as their proofs are still proven to be error-prone [21, 64]. This can be well illustrated by the famous Needham-Schroeder public-key protocol [94], in which a security flaw was found by Gavin Lowe 17 years after its publication [83]. Therefore, automatic verification is very helpful for ensuring the correctness or finding attack on security protocols. In this chapter, we thoroughly perform a formal analysis for the electric vehicle charging protocol [80]. As many symbolic tools, such as Scyther [48], Tamarin [89], ProVerif [29], StateVerif [13], and Athena [113] have been developed for automatic analysis of security protocols using different approaches, selecting the right techniques and tools to verify a complex protocol such as [80] is non-trivial. Due to the number of security and privacy properties claimed by the protocol, no single protocol verifier could give a complete verification of the protocol at present. Thus, we combine the verification capacities from Tamarin and ProVerif, to give a thorough verification of the protocol: Tamarin [89] can handle stateful protocols naturally and allows us to check event order related secrecy and authentication properties, while ProVerif [29] can check observational equivalence [32] so that we use it for checking privacy properties.

Our contributions. First, we present a study of a few state-of-the-art symbolic tools for security protocol analysis and discuss their strength and weakness. We then propose to combine the verification capacities of Tamarin and ProVerif to analyze the electric vehicle

the infinite execution trace involved.

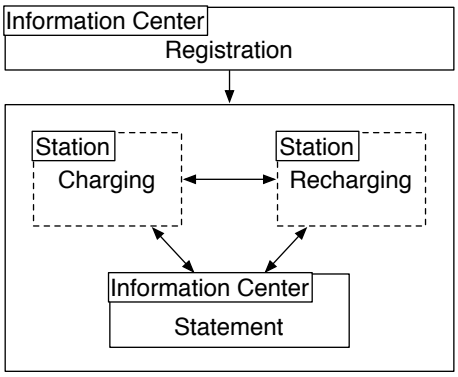


FIGURE 2.1: Protocol Overview

charging protocol, in which secrecy, authentication and privacy properties all play important roles. To the best of our knowledge, this is the first work to use Tamarin and ProVerif for analyzing a single protocol in a compositional approach. Tamarin is used to check event order related secrecy and authentication properties and ProVerif is used to check privacy related properties. We formally define the properties claimed by the protocol and give their verification results in Tamarin and ProVerif.

Structure of the chapter. In Section 2.2, we present the structural overview for this protocol, the cryptographic primitives used in the protocol and the properties required by this protocol. We then describe and compare four state-of-the-art security protocol verifiers in details and address the reasons why we choose ProVerif and Tamarin as the verification tools for this protocol in Section 2.3. In Section 2.4, we will illustrate the modeling in Tamarin along with the verification for secrecy and authentication properties. In Section 2.5, we will show how we specify protocol in ProVerif as well as the anonymity and privacy properties checked. Finally, we draw conclusions in Section 2.6

2.2 The Protocol

In this chapter, we perform formal verification for the properties claimed by an electronic vehicle charging protocol [80]. This protocol is designed to protect the users' privacy,

e.g., anonymity and unlinkability, against the supplier even when they frequently charge and recharge at the stations. It consists of four sub-routines such as registration, charging, recharging and statement, whose overall structure[2] is shown in Figure 2.1.

Registration. Firstly, the users need to register at the information center. They show their identities to the supplier and pay a default deposit to open their accounts. The supplier then give each user a token in return with his balance embedded inside.

Charging and Recharging. During the charging and recharging phases, the users provide their token to the station in an anonymous way. After checking the token, the station will update the token and send it back to the user. Since the station can record the information he receives from the users, the charging and recharging transactions should be taken in a partially blind way.

Statement. When the balance in the user's account is running low, the users could go to the information center again to disclose their identities and top-up money directly into their accounts.

2.2.1 Cryptographic Primitives

To achieve the desired properties, various cryptographic primitives have been used in the protocol, including commitment, zero knowledge proof, and BBS+ signature combined with bilinear pairing.

Commitment. This protocol used the well known commitment scheme developed by Pedersen [96], in which a secret opening value t is chosen randomly by the committer to compute the commitment c over a tuple of public values $(x_0, x_1, \ldots, x_n)$ as $C(x_0, x_1, \ldots, x_n, t) = g_0^{x_0} g_1^{x_1} \ldots g_n^{x_n} g_{n+1}^{t}$. t is unknown to the public at first. When the committer wants to prove that c is the commitment for $(x_0, x_1, \ldots, x_n)$ he made, he could reveal the opening value t

[2] The dash rectangles represent the anonymous charging and recharging operations taking place at the stations.

so that everyone can test if the following equation holds

$$C(x_0, x_1, \ldots, x_n, t) \stackrel{?}{=} c.$$

It has been proven based on certain assumptions [92] that given a commitment c, no polynomial algorithm exists to find the opening value t effectively with respect to $\langle x_0, x_1, \ldots, x_n \rangle$, if $\langle x_0, x_1, \ldots, x_n \rangle$ and t are large random numbers. It has been proven that given a commitment c, no polynomial algorithm exists to find the opening value t with respect to $(x_0, x_1, \ldots, x_n)$, which means only the committer could open the commitment c. Since the commitment scheme is homomorphic, the multiplication of two commitments will give a new commitment with its opening value that equals to the sum of the openings from the formers. In this protocol, the commitment is constructed on the basis of four independent generators in a cyclic group $\mathbb{G}$ denoted as $g_1, g_2, g_3, g_4 \in \mathbb{G}$ such that:

$$C(x_0, x_1, x_2, t) = {g_0}^{x_0} {g_1}^{x_1} {g_2}^{x_2} {g_3}^{t}$$

Zero-Knowledge Proof. As the protocol needs to preserve the privacy properties, the user could not send the signatures, balances and etc. to the supplier directly. As a result, several zero-knowledge proofs are used in this protocol. Zero knowledge proof schemes generally could be represented by $PK\{(s_0, \ldots, s_n) : f_0 \wedge \ldots \wedge f_m\}$ where PK is the proof given, $(s_0, \ldots, s_n)$ are the private informations and $f_0 \ldots f_m$ are the targets the prover wants to prove. As we treat the zero knowledge proofs as black boxes in the verification, we modeled them as functions with two parts of information. One is the public information pub and another is the private information pvt. The prover generates the proof based on both of the public and private information denoted as $prf(pub, pvt)$. To check the proof, the verifier feeds the public information and the proof into the verification function in the form of

$$verif(pub, prf(pub, pvt)) = true,$$

which will return true only when they are correctly matched. Because the prover could give the proof only if he knows the private information, the verifier could then be sure that the prover has the private information he claimed.

BBS+ Signature with Bilinear Pairing. In this protocol, the *BBS+ signature* proposed by Au et al. [16] is employed. It uses an additional generator $g \in \mathbb{G}$. The *BBS+ signature* allows the signer to produce signature in a partially blind way. Specifically, the signer could compute a signature denoted as $A(c, e, r)$ over a commitment c without knowing the values encoded in c by

$$A(c, e, r) = (g * c)^{\frac{1}{e+r}}, \tag{2.1}$$

where e is a fresh random number, r is the long-term private signing key of the signer, and $w = g^r$ is the respective long-term public verification key. To verify the signature, a bilinear pairing function $\hat{e}$ is employed, satisfying that $\hat{e}(g^a, h^b) = \hat{e}(g, h)^{ab}$. As a result, everyone could verify the signature by testing

$$\hat{e}(A(c, e, r), w * g^e) \stackrel{?}{=} \hat{e}(g * c, g). \tag{2.2}$$

2.2.2 Protocol Overview

We give detailed descriptions for the four sub-routines of the protocol as follows.

Registration. The users need to create their accounts at the information center, before they can charge their cars in the V2G system. At the information center, each user discloses his identity I and pays a fixed deposit D to the supplier. Additionally, he freshly chooses random numbers y', s and sends $c_0 = g_0^{y'} g_3^s$ along with a zero knowledge proof $p = PK_1\{(y', s) : c_0 = g_0^{y'} g_3^s\}$ to the supplier. After receiving I, c_0 and p, the supplier checks the identity and verifies the proof p and computes $A = (c_0 g g_0^{y''} g_1^I g_2^D)^{\frac{1}{e+r}}$ according to (2.1) where y'' and e are fresh random numbers. Since the opening of $g_0^{y''} g_1^I g_2^D$ is 0, the opening of the commitment encoded in A is the same as of c_0. When the user gets A, y'' and e from the supplier, he could verify the signature using (2.2) with $c = c_0 g_0^{y''} g_1^I g_2^D$. If

the signature is verified successfully, the user stores the tuple $(A, e, y' + y'', I, D, s)$ for later operations.

Charging. Charging operations are conducted at the stations with users' privacy preserved. Initially, the user has the tuple $(\tilde{A}, \tilde{e}, \tilde{y}, I, \tilde{B}, \tilde{s})$ stored in his device. Firstly, the user randomly picks y' and s to compute the commitment $c_0 = g_0^{y'} g_1^I g_2^{\tilde{B}} g_3^s$ and sends it as well as $\tilde{s}$ to the supplier together with a zero knowledge proof $p = PK2\{(\tilde{A}, \tilde{e}, \tilde{y}, I, \tilde{B}, y', s) : c_0 = g_0^{y'} g_1^I g_2^{\tilde{B}} g_3^s \wedge \hat{s}(\tilde{A}, wg^{\tilde{e}}) = \hat{e}(gg_0^{\tilde{y}} g_1^I g_2^{\tilde{B}} g_3^{\tilde{s}}, g) \wedge D \geq \tilde{B} - v \geq 0\}$. After receiving c_0, $\tilde{s}$ and p, the supplier checks that $\tilde{s}$ has never been used and the proof is correct. If the checking is passed, the supplier picks random numbers y'' and e and computes the signature $A = (c_0 g g_0^{y''} g_2^{-v})^{\frac{1}{e+r}}$ where v is the amount of electricity charged. As can be seen, the new commitment encoded in A equals to $g_0^{y'+y''} g_1^I g_2^{\tilde{B}-v} g_3^s$, so the balance is updated to $\tilde{B} - v$ and the opening is unchanged. The supplier then sends A to the user with y'' and e. When the user receives the A, y'' and e from the supplier, he verifies the correctness of A by (2.2) with $c = c_0 g_0^{y''} g_2^{-v}$ and updates his internal state to $(A, e, y' + y'', I, \tilde{B} - v, s)$.

Recharging. Recharging operation is the same as charging operation except that the updated balance is increased rather than decreased. This could be achieved by multiplying a commitment with a positive balance, which means that the supplier will compute the commitment as $c = c_0 * g_0^{y''} g_2^{+v}$ to increase the balance by v. In addition, the zero knowledge proof $PK3$ in Recharging phase is also the same as $PK2$ except that the balance checking is rewritten into $D \geq \tilde{B} + v \geq 0$.

Statement. When the user wants to increase the balance in his account, the user could go to the information center and top-up money into his account so that the balance in this account could be reset to the default deposit D. If the state stored in the user's device is $(\tilde{A}, \tilde{e}, \tilde{y}, I, \tilde{B}, \tilde{s})$, he needs to disclose his identity I and pay $D - \tilde{B}$ to the supplier. Besides, he randomly picks two numbers y' and s, computes and sends $c_0 = g_0^{y'} g_3^s$ to the supplier with $\tilde{s}$, I, $\tilde{B}$ and a zero knowledge proof $p = PK4\{(\tilde{A}, \tilde{e}, \tilde{y}, y', s) : c_0 =$

$g_0^{y'} g_3^s \wedge \hat{s}(\tilde{A}, wg^{\tilde{e}}) = \hat{e}(gg_0^{\tilde{y}} g_1^I g_2^{\tilde{B}} g_3^{\tilde{s}}, g)\}$. If $\tilde{s}$ has not been used before, the supplier checks the correctness of the proof p and computes the signature $A = (c_0 g g_0^{y''} g_1^I g_2^D)^{\frac{1}{e+r}}$ where y'' and e are freshly generated random numbers. Then, the supplier will send it with y'' and e to the user. After checking the signature with (2.2) where $c = c_0 g_0^{y''} g_1^I g_2^D$, the user updates the tuple in his device as $(A, e, y' + y'', I, D, s)$ and completes the statement.

2.2.3 Assumptions

Complex cryptographic primitives are used in the protocol. Additionally, the protocol requires redundancy checking, infinite set maintenance and algebra calculation. Modeling these complex operations could easily lead to non-termination of the verification process. Thus we make some assumptions below.

The cryptographic primitives are prefect. The adversary cannot guess the correct opening value or forge another opening value to open the commitment. For zero-knowledge proofs, we assume that they will not cause any information leakage for the secret values and they can prove what they intended to prove. For BBS+ signature, signature could not be forged without the signing key.

The supplier will not accept a same opening value twice. This assumption ensures that no user could use the opening value and the signature issued from the supplier twice. The reasons are as follows. The zero-knowledge proofs make sure that the opening value is embedded in the commitment. Besides, the commitment scheme requires that no other opening value could be forged. When the supplier receives a non-duplicated opening value, if the verification is passed, the opening value and the signature should never be used before.

Algebraic addition operation on random numbers never introduce duplicated values. In the protocol, add function is used to compute the sum of two numbers, which is used in cryptos such as BBS+ signature and commitment. In symbolic verification, we say two

terms are equal when they are structurally equivalent. Since we only apply add function to random numbers in the protocol, if algebraic operations on random numbers never introduce duplicated values, two add results should be equal whenever they are structurally equivalent, which ensures the correctness of the symbolic verification.

Balance bound checking in zero knowledge proofs are not considered. During the charging and recharging, balance checking is implicitly checked in the zero knowledge proofs $PK2$ and $PK3$. However, we omit the balance bound checking in ***PK2*** and ***PK3*** during the verification, since $<$ and $\leq$ conditions are not supported in ProVerif and Tamarin.

2.2.4 Primitives Modeling

According to the applications of the primitives and the structure of the protocol, we modeled the primitives as follows:

Commitment. Two forms of commitments are generated in the protocol such as $g_0^y g_3^s$ and $g_0^y g_1^I g_2^B g_3^s$. We modeled them as $resC(y, s)$ and $chtC(y, I, B, s)$ in the tools. When the supplier receives them, he computes the signature A accordingly in a consistent representation.

Zero Knowledge Proof. In this protocol, four zero knowledge proofs are used such as $PK1$, $PK2$, $PK3$ and $PK4$. Since the balance bound checking is omitted in $PK2$ and $PK3$, they become identical so we merged them into one zero knowledge proof of $PK23$. For $PK1$, the prover provides a proof knowledge $regZK(c_0, y', s)$ with a verification function to check the correctness of c_0

$$PK1(resC(y', s), regZK(resC(y', s), y', s)) = true.$$

In $PK23$, the prover's secrets are $(\tilde{A}, \tilde{e}, \tilde{y}, I, \tilde{B}, y', s)$ and the known information to the supplier are $(c_0, \tilde{s}, r)$, so he provide a proof of $chtZK(c_0, \tilde{A}, \tilde{e}, \tilde{y}, I, \tilde{B}, y', s)$ along with a

verification function

$$PK23(chtC(y', I, \tilde{B}, s), \tilde{s}, r, chtZK(chtC(y', I, \tilde{B}, s),$$

$$sysA(\tilde{y}, I, \tilde{B}, \tilde{s}, \tilde{e}, r), \tilde{e}, \tilde{y}, I, \tilde{B}, y', s)) = true,$$

which checks if the c_0 and $\tilde{A}$ are correctly formed with respect to $(\tilde{e}, \tilde{y}, I, \tilde{B}, y', s)$. For statement sub-routine, the proof knowledge $zk = stmZK(c_0, \tilde{A}, \tilde{e}, \tilde{y}, y', s)$. The verifier check the zk with $PK4$ to ensure the c_0 and $\tilde{A}$ are correct.

$$PK4(resC(y', s), \tilde{s}, I, \tilde{B}, r, stmZK(resC(y', s),$$

$$sysA(\tilde{y}, I, \tilde{B}, \tilde{s}, \tilde{e}, r), \tilde{e}, \tilde{y}, y', s)) = true.$$

BBS+ signature with bilinear pairing. We model the BBS+ signatures in a systematic form as $sysA(y, I, B, s, e, r)$ in which y is the addition of two nonces, I is the identity of the user, B is the current balance in his account, s is the opening of the commitment encoded in the signature, e is the random number chosen by the supplier and r is the private signing key of the supplier. wg^e is modeled by $compose(sysgr(r), e) = sysger(e, r)$ and $w = sysgr(r)$ is a public information. We defined an $extract$ function to model the behavior of (2.2) as

$$extract(sysA(y, I, D, s, e, r), sysger(e, r))$$

$$= syse(sysall(y, I, D, s))$$

in which the e and r is eliminated because of the bilinear function is used. Thus the user could compute the value of $syse(sysall(y, I, D, s))$ directly.

2.3 Tools

Many symbolic tools, such as Scyther [48], Tamarin [89], ProVerif [29], StatVerif [13] have been developed for automatic analysis of security protocols using several approaches. They have different capabilities for analyzing different protocols with respect to different properties.

Scyther [48] is a tool based on the strand space [60] and the Athena [113] but extends them with trace patterns to reduce the search space. Although unbounded verification could be achieved by Scyther for some protocols, Scyther sometimes bounds the session number to ensure termination of the verification. Additionally, stateful protocol verification and privacy properties are not supported in Scyther.

Tamarin [89, 104] uses multiset rewriting [42] to specify the adversary's capabilities together with a guarded fragment [11] of first-order logic for security properties and equational theories for algebraic properties. Due to the fact that multiset rewriting rules can be directly specified in a model to represent the execution state of the protocol, Tamarin becomes a powerful tool for verifying stateful protocols. Additionally, session indexes can be specified in the query so that authentication properties and secrecy properties with event ordering could be checked in Tamarin.

ProVerif [29] is developed actively since 2001, which uses Horn clauses to represent the adversary's capability and backward deduction to check for secrecy. Over approximation on generated session nonces is deployed to limit the searching space but also leads to false attacks. By combining the secrecy checking with inserting special events which indicates the begin and end of the protocol execution [30], authentication checking is then allowed in ProVerif. Blanchet et al. later extended ProVerif with observational equivalence checking [32] and strong secrecy checking [31] so that privacy leakage can be found in protocols.

StatVerif [13] later extends ProVerif with the global mutable state. It could handle protocol with explicit global state by converting the processes into a set of clauses upon which ProVerif could verify. In the meanwhile, further abstractions are needed for verifying protocol with infinite state spaces.

Comparison. The differences of these tools are summarized in Table 2.1. As the electric charging protocol [80] analyzed in this chapter is a stateful protocol and it uses the commit-

Property	Scyther	Tamarin	ProVerif	StatVerif
Secrecy	UB/B	UB	UB	UB
Authentication	UB/B	UB	UB	N.A.
Stateful Verification	N.A.	Infinite	Weak	Explicit
Explicit Event Index	N.A.	Supported	N.A.	N.A.
Strong Secrecy	N.A.	N.A.	Supported	N.A.
Observational Equivalence	N.A.	N.A.	Supported	N.A.

Table 2.1: Tool Comparison. (UB : Unbounded; B : Bounded; N.A. : Not Available.)

ment scheme which requires explicit event index ordering, Tamarin is the best candidate for analyzing secrecy and authentication properties. On the other hand, strong secrecy and observational equivalence are required for checking privacy properties in this protocol. ProVerif is the only tool that supports these features. Thus, we integrate the verification capacities from Tamarin and ProVerif to give a thorough verification of the protocol.

2.4 Analysis in Tamarin

2.4.1 Abstractions

We abstract the original protocol to ensure the termination of the verification process in Tamarin. Since protocol abstraction can only introduce false alarms, if the protocol is proven as secure after abstraction, its original version should be secure as well.

Fixing the balance for each sub-routines. During the verification in Tamarin, if the balance is allowed to be increased and decreased in the protocol, verification procedure will try to increase its value by infinite times without termination. Because the balance value is not relevant to the secrecy and authentication properties considered in the verification, we fixed the balance along the protocol execution. Since the balance is abstracted to a fixed value, the charging and recharging behavior are then identical. So we merged them into one operation denoted as cht in the model.

Setting the value y'' to 0. In the protocol, y'' is chosen randomly in every sub-routine by

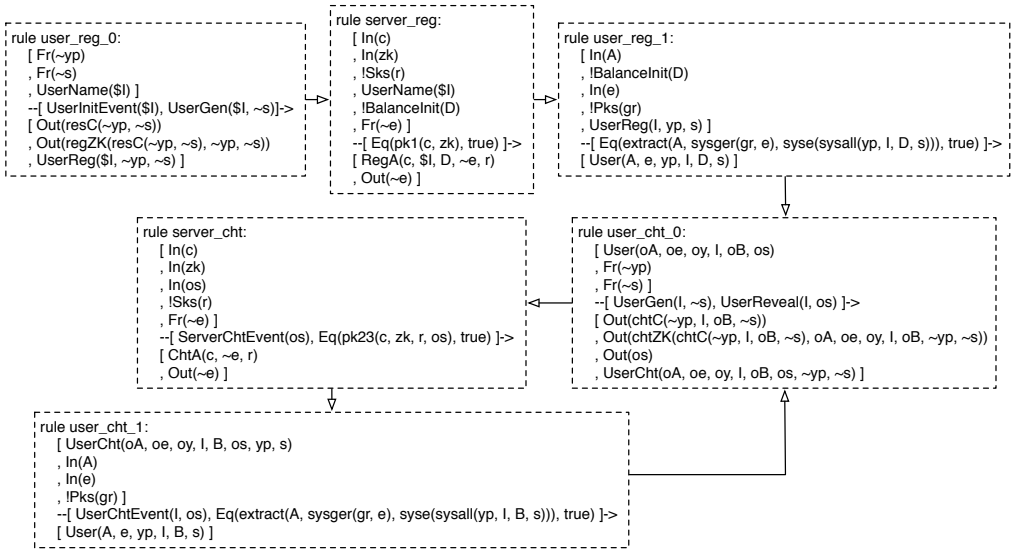


FIGURE 2.2: Modeling in Tamarin

the supplier. When the supplier computes the signature, he adds y'' to y' which is encoded in the commitment generated by the user. Since the add function is communicative, we need to reflect the algebra property for add in the model. In the meanwhile, the multiset rewriting rules can only apply to a whole message, so we have to rewrite all the messages where the add function may appear. As a result, the verification process could not terminate because of the complexity introduced by the duplicated rules specified in the model. In order to make the model terminable, we set the y'' generated by the supplier to 0. Thus $add(y', y'')$ is equivalent to y' and modeling the behavior of the algebra function add becomes unnecessary. In fact, y'' is a public value as the supplier will send it out in every session, so fixing its value will not affect the verification results for secrecy and authentication checking.

2.4.2 Modeling

Multiset rewriting rules are specified in Tamarin to model the protocol execution. For each sub-routine described in Section 2.2.2, we divide the user's part into two phases. One for sending out the commitment and zero-knowledge proofs, and the other for verifying the signature and updating the internal state of the user. After the registration, every user

maintains an internal state of $User(A, e, y, I, B, s)$ in which A is the signature issued from supplier, e and y are nonces, I is the identity of the user, B is the balance in the user's account and s is the opening for the user's commitment encoded in A. Registration and charging/recharging behaviors for both participants are shown in Figure 2.2. We use *user_reg_0* and *user_reg_1* to represent the user actions during the registration, and use *server_reg* to specify the server's registration behavior. User states are maintained for all of the sub-routines. The two user actions in the registration phase are linked by the user state *UserReg*. Charging/recharging and statement phases have similar structures. For instance, in the charging phase, we use *user_crg_0* and *user_crg_1* to represent the user actions during the charging, and use *server_crg* to specify the server's charging behavior. As can be seen from the figure, a loop exists in the user charging/recharging phase, which means that a user could do charging/recharging for infinite times. In Tamarin, $pk1$, $pk23$, $pk4$ and signature checking are modeled according to Section 2.2.4. In order to check the zero knowledge proofs and signatures, we defined an axiom on function Eq to test the equivalence of two terms

$$Eq(x, y) \Rightarrow x = y.$$

Thus $Eq(pk1(c, zk), true)$ could be tested along the execution. The models are available in [1].

2.4.3 Checking Secrecy and Authentication

Secrecy. One interesting secrecy property required by the electric charging protocol is the conditional secrecy property for the opening value s of the commitment. As the user explicitly sends the opening s out for verification, s will be known to the public. In the meanwhile, s should be kept secret before the user intends to do so.

Definition 2.1. ***(Commitment secrecy).*** *A nonce s satisfies* commitment secrecy *with respect to the event $open$ if and only if s is secret before event $open$ is engaged.*

This property can be specified in Tamarin as follows:

$$\forall I, s, i, j. generate(I, s)@i \wedge know(s)@j \\ \Longrightarrow (\exists r. open(I, s)@r \wedge r < j)$$

in which i, j and r are session ids, I is the identity of the user, and s is the value that should satisfy *commitment secrecy* regarding to $open$ event. The formula means whenever the opening value s generated by I is known to the adversary, there exists an event $open$ explicitly engaged by I before it is known.

Authentication. As this protocol is a charging protocol, we need to make sure that the correspondences between the users and the supplier are established correctly. We adopt the definition of non-injective agreement and agreement from [85] and formalize them for the protocol in Tamarin. In the protocol, because we deem the supplier as the adversary, the users need to make sure that the other participant of the protocol is taking the role of supplier. Thus, we formalize the non-injective agreement between the supplier and the user as

$$\forall I, s, i. UserCht(I, s)@i \Longrightarrow (\exists r. SupplierCht(s)@r)$$

which means whenever a user is taking his role for charging or recharging, the supplier is also taking his role in the protocol. Additionally, we formalize the injective agreement as

$$\forall I, s, i. UserCht(I, s)@i \Longrightarrow (\exists r. SupplierCht(s)@r \\ \wedge (\forall j. UserCht(I, s)@j \Rightarrow i = j)).$$

The injective agreement makes sure that for any operation taken by the supplier, there is only one corresponding user.

Verification results. We have checked the above properties against different scenarios of the protocol, the verification results are summarized in Table 2.2. All the experiments are conducted under Mac OS X 10.9.1 with 2.3 GHz Intel Core i5 and 16G 1333MHz DDR3.

Routine	Property	Result	Time	Step
Charging and Recharging	Secrecy	SAT	1.72s	54
	Non-injective Auth	SAT	7.14s	231
	Injective Auth	SAT	1092.87s	23895
Statement	Secrecy	SAT	1.12s	20
	Non-injective Auth	SAT	1.56s	33
	Injective Auth	SAT	9.09s	251
Charging, Recharging and Statement	Secrecy	N.T.	-	-
	Non-injective Auth	N.T.	-	-
	Injective Auth	N.T.	-	-

Table 2.2: Verification results for Tamarin : SAT - Satisfied, N.T. - Non-terminating

After the registration is finished, any user could do charging/recharging for infinite times with these three properties being preserved. In addition, these properties are also hold for infinite times of statement operations after registration. However, if we check the properties with the charging/recharging and statement sub-routines combined, the verification procedure does not terminate.

2.5 Analysis in ProVerif

2.5.1 Abstractions

During the protocol modeling in ProVerif, some abstractions are made to ensure the termination of the verification process.

Fixing the values of y'' and e. In ProVerif, the value of a newly generated nonce depends on two factors: the name of the nonce and the value of the messages received before the generation point. In Section 2.2.2, we have shown that two nonces y'' and e will be generated by the supplier in every session after the supplier receives the commitment, the zero-knowledge proof and an opening value $\tilde{s}$. Additionally, the nonces $\tilde{y}''$ and $\tilde{e}$ generated in the last session are encoded in the zero-knowledge proof, which then leads to the infinite dependency traces of y'' and e. Thus, the verification process cannot terminate. To break the infinite dependency chain, we model y'' and e as two globally shared nonces instead

of generating them freshly in every session. Because fixing the value of y'' and e will not affect the users' privacy, this abstraction will not introduce false positives into the verification result.

Setting the value of y'' to 0 for intractability and unlinkability checking. During intractability and unlinkability analysis, the communicative law required by the function add will lead to the non-termination of the verification. Since we could not remove the communicative law which may lead to false negatives, we set $y'' = 0$ so that $add(y', y'')$ is equivalent to y'. Then we could safely eliminate the usage of add with its functionality preserved. As the users' privacy will not be weaken by fixing y'', this abstraction will not introduce any false positives either.

2.5.2 Modeling

During the modeling in ProVerif, locations are modeled by public channels and the users' state are passed by the process arguments to simulate the protocol execution. We have modeled eight processes for four communication sub-routines between the users and the supplier by following the protocol specification described in Section 2.2.2. Since the infinite iterations cannot be specified in ProVerif model, we explicitly call different phases when the current routine is finished. For instance, when we check location privacy, we call $UserCrg$ by passing the user state as arguments to the process. (See the models in [1].)

2.5.3 Checking Privacy

Users' privacy is the main purpose of designing this protocol, in which the supplier is the potential adversary for breaking the users' privacy. In this section, we give precise definitions of anonymity and location privacy for the protocol, and we investigate other properties for the protocol as well, including intractability, anonymity, strong anonymity and unlinkability. We present the verification results for all of them. Privacy properties are normally modeled by observational equivalence in the applied pi calculus, which is

a widely accepted approach [53, 12, 57]. In the following discussions, we use ECP to represent the well-formed representation [12] for the electric vehicle charging protocol and use Reg, Crg, Rcg and Stm for registration, charging, recharging and statement subroutines respectively. $Rtn\{n_1/p_1, \ldots, n_m/p_m\}$ means that the routine Rtn parameterized with $p_1, \ldots, p_m$ is instantiated by the values $n_1, \ldots, n_m$. Since the protocol is stateful, $Reg.Crg.Crg$ is different from $Reg.(Crg|Crg)$. Additionally, infinite iterations of the process cannot be specified in ProVerif, so we only give proofs to finite execution iterations with infinite replications and define ECP_{Reg} and ECP_{Crg} as follows

$$ECP_{Reg} = !v(i).Reg\{i/id\},$$
$$ECP_{Crg} = !v(i).Reg\{i/id\}.Crg\{i/id\}.$$

Similarly, we could define the well-formed protocol for recharging as ECP_{Rcg}.

Location privacy. Location privacy should be guaranteed for the users' behaviors in the stations, which could be specified as

Definition 2.2. ***(Location privacy).*** *A user A's charging behavior conducted at station X satisfies location privacy if there exists a user B at station Y s.t.*

$$\begin{aligned} & C[(Reg\{A/id\}.Crg\{A/id, X/l\} \\ & \quad |Reg\{B/id\}.Crg\{B/id, Y/l\})] \\ \sim\ & C[(Reg\{A/id\}.Crg\{A/id, Y/l\} \\ & \quad |Reg\{B/id\}.Crg\{B/id, X/l\})]. \end{aligned}$$

The stations could be modeled in ProVerif by different channels. Location privacy could also be defined for recharging behaviors similarly. In addition, we also specify *intractability* for users' behaviors as

Definition 2.3. ***(Intractability).*** *Two subsequent charging behaviors conducted by a user*

A at station X satisfies intractability if there exists a user B and a station Y s.t.

$$\begin{aligned} & C[(Reg\{A/id\}.Crg\{A/id, X/l\}.Crg\{A/id, X/l\} \\ & \quad |Reg\{B/id\}.Crg\{B/id, Y/l\}.Crg\{B/id, Y/l\})] \\ \sim\ & C[(Reg\{A/id\}.Crg\{A/id, X/l\}.Crg\{A/id, Y/l\} \\ & \quad |Reg\{B/id\}.Crg\{B/id, Y/l\}.Crg\{B/id, X/l\})] \end{aligned}$$

which means the adversary cannot distinguish if a user performs charging at a same station or different stations. Similarly, we can define intractability for recharging behaviors.

Anonymity and strong anonymity. Anonymity should be preserved when the user is doing the charging operation and the recharging operation at stations. Anonymity for charging is defined as follows.

Definition 2.4. ***(Anonymity).*** *A well formed protocol ECP_{Crg} satisfies anonymity property for a user A's charging behavior if there exists a user B s.t.*

$$\begin{aligned} & C[ECP_{Crg}|(Reg\{A/id\}|Reg\{B/id\}).Crg\{A/id\}] \\ \sim\ & C[ECP_{Crg}|(Reg\{A/id\}|Reg\{B/id\}).Crg\{B/id\}] \end{aligned}$$

A stronger notion for anonymity is proposed in [12] which ensures that the adversary cannot tell whether a user A has participated a protocol run or not.

Definition 2.5. ***(Strong anonymity).*** *A well formed protocol ECP_{Crg} satisfies strong anonymity property for a user A's charging behavior if*

$$\begin{aligned} & C[ECP_{Crg}|Reg\{A/id\}] \\ \sim\ & C[ECP_{Crg}|Reg\{A/id\}.Crg\{A/id\}] \end{aligned}$$

Similarly, we could define (strong) anonymity for recharging behaviors.

Unlinkability. In protocol ECP, unlinkability is claimed for charging and recharging operations at stations so that the no adversary, including the supplier, could tell that two operations are initiated by the same user.

Routine	Property	Result	Time
Charging	Location Privacy	SAT	131.64s
	Intractability	SAT	13.87s
	Anonymity	SAT	1391.20s
	Strong Anonymity	SAT	1717.25s
	Unlinkability	SAT	5.94s
Recharging	Location Privacy	SAT	132.43s
	Intractability	SAT	14.96s
	Anonymity	SAT	1372.65s
	Strong Anonymity	SAT	1804.23s
	Unlinkability	SAT	6.63s

Table 2.3: Verification results for ProVerif : SAT - Satisfied

Definition 2.6. ***(Unlinkability).*** *A well formed protocol ECP_{Crg} satisfies unlinkability for a user A's charging behavior if there exists a user B s.t.*

$$C[(Reg\{A/id\}.Crg\{A/id\}.Crg\{A/id\})|(Reg\{B/id\})]$$
$$\sim C[(Reg\{A/id\}.Crg\{A/id\})|(Reg\{B/id\}.Crg\{B/id\})]$$

Verification results. We have successfully verified location privacy, intractability, anonymity, strong anonymity and unlinkability for users' charging and recharging behaviors in ProVerif. We do the experiments under Mac OS X 10.9.1 with 2.3 GHz Intel Core i5 and 16G 1333MHz DDR3. The verification results are summarized in Table 2.3.

2.6 Discussions

In this chapter, we presented the verification of an electric vehicle charging protocol proposed by Liu et al. [80] using two most efficient tools. We have checked various security and privacy properties for the protocol, such as secrecy and authentication in Tamarin, and location privacy, intractability, anonymity and unlinkability in ProVerif. Moreover, we have addressed the capabilities of Tamarin and ProVerif. We are the first to combine their symbolic verification results for a single protocol.

During the verification, we find that Horn logic used by ProVerif is generally more efficient than the multiset rewriting logic adopted in Tamarin. Furthermore, because the multiset rewriting is very powerful as a specification language, where the protocol communications and states can be explicitly specified, a single protocol could be modeled in many different ways. Thus, it is hard for its users to choose the right modeling method, as some of the modeling methods might result in non-termination of the verification. Even though the applied π-calculus with protocol states [73] has been developed to model the (stateful) protocols in Tamarin, it still needs its users to manually write theorems based on the machine generated multiset rewriting rules, which makes the modeling even more laborious. On the other hand, security protocols can be modeled by the Horn logic in ProVerif using the applied π-calculus. Moreover, Horn logic can be translated from the applied π-calculus automatically. Hence, we conclude that Horn logic is more user-friendly as a specification language comparing with multiset rewriting logic.

However, in order to help the termination of the verification in ProVerif [29] using Horn logic, Blanchet [29] introduces an abstraction that merges the nonces in different sessions when the nonces have the same name and the sessions have the same trace. This abstraction can generally lead to false alarms to the protocol verification [77, 78] when the commitment schemes [96] or the timing constraints [35, 84, 54] are involved. For instance, consider the following process, where s is a secret constant.

$$P = \ !\nu n.c(x).\bar{c}(n).\text{if } x = n \text{ then } c(s).0$$

P generates a nonce n, receives a value x from network and then outputs the nonce n to the network. If x is equal to n, P then sends out the secret s. Since the nonce n is a random number that is unknown to the public until it is revealed, the value x can never equal to n. So the secrecy property of s should be preserved. However, according to [45], ProVerif reports false alarms because of its abstraction of nonces. We thus propose a verification framework, in Chapter 3, based on Horn logic, which requires no abstraction during the

verification. Hence, our framework facilitates the verification of security protocols, involving time and commitment scheme. By using the framework provided in Chapter 3, we can successfully verify the commitment secrecy of the electric vehicle charing protocol in 42 milliseconds.

Chapter 3

Timed Security Protocol Verification

Quantitative timing is often relevant to the security of systems, like web applications, cyber-physical systems, etc. Verifying timed security protocols is however challenging as both arbitrary attacking behaviors and quantitative timing may lead to undecidability. In this chapter, we propose a service framework to support intuitive modeling of the timed protocol, as well as automatic verification of an unbounded number of protocol sessions. The partial soundness and completeness of our verification algorithms are formally proved. The evaluation results show that our approach is efficient and effective in both finding security flaws and giving proofs.

3.1 Introduction

Timed security protocols are used extensively. Many security applications [103, 38, 23] use time to guarantee the freshness of messages received over the network. In these applications, messages are associated with timing constraints so that they can only be accepted in a predefined time window. As a result, relaying and replaying messages are allowed only in a timely fashion. It is known that security protocols and their manual proofs are

error-prone, which has been evidenced by multiple flaws found in existing proved protocols [105, 100, 61, 20, 19, 18]. It is therefore important to have automatic tools to formally verify these protocols.

However, existing methods and tools for security protocol verification often abstract timestamps away by replacing them with nonces. The main reason is that most of the decidability results are given for untimed protocols [87, 101]. Thus, the state-of-the-art security protocol verifiers, e.g., ProVerif [29], Athena [113], Scyther [48] and Tamarin [89], are not designed to specify and verify time sensitive cryptographic protocols. Abstracting time away may lead to several problems. First, since the timestamps are abstracted as nonces, the message freshness checking in the protocol cannot be correctly specified. As a consequence, attacks found in the verification may be false alarms because they could be impractical when the timestamps are checked. Second, omitting the timestamp checking could also result in missing attacks. For instance, the timed authentication property ensures the satisfaction of the timing constraints in addition to the establishment of the event correspondence. Without considering the timing constraints, even though the agreement is verified under the untimed configuration correctly, the protocol may still be vulnerable to timing attacks. Third, with light-weight encryption, which are often employed in cyber-physical systems, it might be possible to decrypt secret messages in a brute-force manner given sufficient time. In applications where long network latency is expected, it is therefore essential to consider timing constraints explicitly and check the feasibility of attacks.

Contributions. In this work, we provide a fully automatic approach to verify timed security protocols with an unbounded number of sessions. Our contributions are fourfold. (1) In order to precisely specify the capabilities of the adversary, we propose a service framework in which the adversary's capabilities are modeled as various services according to the protocol specification and cryptographic primitives. Thus, when the protocol is vul-

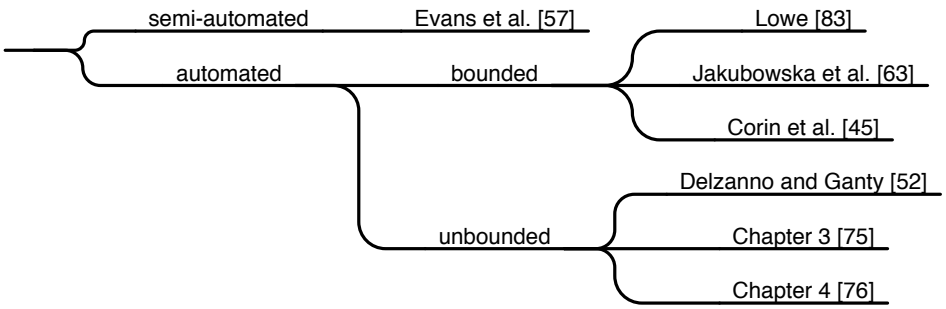


FIGURE 3.1: Roadmap for Related Works

nerable, there should exist an attack trace consisting of the services in a certain sequence. (2) An automatic algorithm is developed in this work to verify the timed authentication properties with an unbounded number of sessions. Since security protocol verification is undecidable in general [41], we cannot guarantee the termination of our algorithm. We thus prove our algorithm as partially sound and complete in Section 3.3. (3) Having time in security protocol verification adds another dimension of complexity. Thus we propose the finite symbolic representation for the timing constraints with approximation. We prove that the protocol is guaranteed to be secure when it is full verified by our algorithm. Additionally, when the protocol specification is in a specific form, we also prove that our algorithm does not introduce false alarms. (4) A verifier named TAuth is developed based on our method. We evaluate TAuth using several timed and untimed security protocols [35, 40, 94, 72, 33, 103]. The experiment results show that our approach is efficient and effective in both finding security flaws and giving proofs.

Related works. The roadmap of related works are shown in Figure 3.1. Evans et al. [59] introduced a semi-automated way to analyze timed security protocols. They modeled the protocols with CSP and checked them with PVS. In [86], Lowe proposed finite state model checking to verify bounded timed authentication. In order to avoid the state space explosion problem, protocol instances and time window are bounded in the verification. Jakubowska et al. [65] and Corin et al. [47] used Timed Automata to specify the protocols

and used Uppaal to give bounded verifications. Our method is different from theirs as our verification algorithm is fully automatic and the verification result is given for an unbounded number of sessions.

The work closest to ours was proposed by Delzanno and Ganty [54] which applies $MSR(\mathcal{L})$ to specify unbounded crypto protocols by combining first order multiset rewriting rules and linear constraints. According to [54], the protocol specification is modified by explicitly encoding an additional timestamp, which represents the protocol initialization time, into some messages. Thus the attack could be found by comparing that timestamp with the original timestamps in the messages. However, it is not clearly illustrated in [54] how their approach can be applied to timed security protocol verification in general. On the other hand, our approach could be directly applied to crypto protocols without any manual modification to the protocol specification.

We adopt the Horn logic which is similar to the one used in ProVerif [29], a very efficient security protocol verifier designed for untimed cryptographic protocol, and extend it with timestamps and timing constraints. However, the extension for time is nontrivial. In ProVerif, the fresh nonces are merged under the same execution trace, which is one of major reasons for its efficiency. When time is involved in the protocol, the generation time of the nonces in the protocol becomes important for the verification. Thus merging the session nonces under the same execution trace often introduces false alarms into the verification results. In order to differentiate the nonces generated in the sessions, we encode the session nonces into the events engaged in the protocol and use the events to distinguish them. Additionally, our approach takes care of the infinite expansion of timing constraints, which is discussed in Section 3.3.1.

Type	Expression	
Timestamp(t)	t	
Message(m)	$g(m_1, m_2, ..., m_n)$	(function)
	$a[]$	(name)
	$[n]$	(nonce)
	v	(variable)
	t	(timestamp)
Event(e)	$\langle m, t \rangle$	(knowledge)
	$e(m_1, m_2, ..., m_n)$	(event)
Constraint(B)	$\mathcal{C}(t_1, t_2, ..., t_n)$	
Service(S)	$[\ G\ ]\ e_1, e_2, ..., e_n \dashv [\ B\ ] \mapsto e$	
Query(Q)	$accept(\ldots) \leftarrow [\ B\ ] \vdash init_1(\ldots), \ldots, init_n(\ldots)$	

Table 3.1: Service Syntax Hierarchy

3.2 Protocol Specification Framework

We introduce the proposed protocol specification framework in this section. In the framework, the security protocols and the cryptographic primitives are modeled as various services accessible to the *Adversary* for conducting attacks. Generally, these services receive inputs from the adversary and send the results back to the adversary as output over the network. Timestamps are tagged to the messages to denote when they are known to the adversary. We assume the adversary model presented in this framework is an active attacker who can intercept all communications, compute new messages and send the messages he obtained. For instance, he can use all the publicly available functions including encryptions, decryptions, concatenations, etc. He can also ask legal protocol participants to take part in the protocol when he needs. Thanks to the introduction of time, key expiration and message compromise can also be specified by adding additional services.

3.2.1 Service Syntax

In our framework, services are represented by a set of Horn logic rules guarded by timing constraints. We adopt the syntax shown in Table 3.1 to define the services. *Messages*

could be defined as *functions*, *names*, *nonces*, *variables* or *timestamps*. *Functions* can be applied to a sequence of *messages*; *names* are globally shared constants; *nonces* are freshly generated values in sessions; *variables* are memory spaces for holding *messages*; and *timestamps* are values extracted from the global clock during the protocol execution. A *event* can be a *message* tagged by a *timestamp* denoted as $\langle m, t \rangle$, which means that the message m is known to the adversary at time t. Otherwise, it is an user-defined *event* in the form of $e(m_1, \ldots, m_n)$ where e is the event name and $m_1, \ldots, m_n$ are the event arguments. The events are used for specifying authentication properties and distinguishing different sessions. B is a set of closed timing constraints assigned on the *timestamp* pairs. Each constraint is in the form of $t - t' \sim d$ where t and t' are *timestamps*, d is an integer constant (∞ is omitted), and $\sim$ denotes either $<$ or $\leq$. We denote the maximum value of d in a timing constraint set B as $max(B)$. For simplicity, when a timing constraint $t - t' \sim d \in B$, we write $d(B, t, t')$ to denote the integer constant d, and $c(B, t, t')$ to denote the comparator $\sim$[1]. G is a set of untimed conditions such as message inequivalence. A *service* $[\ G\ ]\ e_1, e_2, ..., e_n \ \dashv[\ B\ ]\!\!\rightarrow\ e$ means that if the *events* $e_1, e_2, ..., e_n$, the conditions G and the constraints B are satisfied, the adversary can invoke this service and obtain e as the result.

3.2.2 Service Modeling

In the following, we show how to model the timed authentication protocols in our framework. We illustrate the service modeling using a simple example called the Wide Mouthed Frog (WMF) protocol [35] as described below.

$$A \rightarrow S : A, \{t_A, B, k\}_{k_A}$$

$$S \rightarrow B : \{t_S, A, k\}_{k_B}$$

[1] If a timing constraint is not specified exactly in this form, it should be possible to change the constraint into this form. For instance, $t - t' > 3$ can be changed into $t' - t < -3$.

In the protocol, A and B are two users *Alice* and *Bob*, and S is a trust server who shares different secret keys with different users. The goal of this protocol is to share a fresh key k from *Alice* to *Bob*. k_A is the secret key shared between server and *Alice*, and k_B is the corresponding secret key for *Bob*. k is a fresh session key generated by *Alice*, which should be different in different sessions. t_A is a timestamp generated by *Alice*. Similarly, t_S is a timestamp generated by the server. In the protocol, we assume that the clock drift for every participants is negligible, so that the message freshness checking is valid during the execution.

When the server receives the request from *Alice*, it checks its freshness by comparing the t_A with the current clock reading t_S. If t_A and t_S satisfy the pre-defined constraint C_1, the server then sends the second message to *Bob*. Upon receiving the message from the server, *Bob* decrypts it and compares t_S with his clock reading t_B. If the timestamp checking C_2 is passed and the message is properly formed, *Bob* then believes that k is a fresh key shared with *Alice*. In fact, there exists an attack [9] to the protocol which is resulted from the symmetric structure of the exchanged messages.

$$
\begin{aligned}
A \rightarrow S : &\qquad A, \{t_A, B, k\}_{k_A} \\
S \rightarrow I(B) : &\qquad \{t_S, A, k\}_{k_B} \\
I(B) \rightarrow S : &\qquad B, \{t_S, A, k\}_{k_B} \\
S \rightarrow I(A) : &\qquad \{t_{S'}, B, k\}_{k_A} \\
I(A) \rightarrow S : &\qquad A, \{t_{S'}, B, k\}_{k_A} \\
S \rightarrow B : &\qquad \{t_{S''}, A, k\}_{k_B}
\end{aligned}
$$

In the attack trace, the adversary I personates *Bob*, hijacks the second message and sends it back to the server within the timing constraint C_1. Then, the server would treat it as a valid request from *Bob* and update the t_S to its current clock reading. By doing this repeatedly, the timestamp in the request can be extended to an arbitrary large value. As a result, when *Bob* receives a message that passes the timestamp checking, the request from

Alice may not be timely any more. Hereafter, we assume that the server and *Bob* check the freshness of the received messages with following timing constraints: $C_1 = t_S - t_A \leq 2$ and $C_2 = t_B - t_S \leq 2$. Notice that in general, the constraints should be set according to the protocol specification, network latency, etc.

Crypto Services. Cryptographic primitives are usually specified as services without network latency. Generally, we have two types of crypto services, which are constructors and destructors. Constructors are used to generate new messages such as concatenation and encryption, whereas destructors are used to extract messages from the constructed messages. For instance, the constructor and the destructor for symmetric encryption can be modeled as follows.

$$\langle m, t_1 \rangle, \langle k, t_2 \rangle \dashv\!\lfloor\, t_1 \leq t \wedge t_2 \leq t \,\rfloor\!\mapsto \langle enc_s(m,k), t \rangle \tag{3.1}$$

$$\langle enc_s(m,k), t_1 \rangle, \langle k, t_2 \rangle \dashv\!\lfloor\, t_1 \leq t \wedge t_2 \leq t \,\rfloor\!\mapsto \langle m, t \rangle \tag{3.2}$$

The service (3.1) means that if the adversary has a message m and a key k, this service can generate the symmetric encryption for m by k, and the timing t of receiving the encryption should be later than the timing t_1 and t_2 when m and k are known to the adversary. The symmetric decryption service is similarly defined in service 3.2.

For some cryptographic primitives, additional constraints can be added for special purposes. For instance, RSA encryption may consume non-negligible time to compute. If the encryption time has a lower bound d, we could use the following constructor to model the additional requirement on time.

$$\langle m, t_1 \rangle, \langle pk, t_2 \rangle \dashv\!\lfloor\, t - t_1 > d, t - t_2 > d \,\rfloor\!\mapsto \langle RSA(m, pk), t \rangle$$

Protocol Services. Protocol services are used to specify the execution of the protocol. These services are directly derived from the protocol specification. Specifically, for the WMF protocol, the server S answers queries from all its users. After receiving a request from a user I, S extracts the message content and checks the timestamp. If the timestamp is generated within 2 time units, S sends out the encryption of an updated timestamp t_S, the initiator's name and the session key k under the responder's shared key. The service

provided by the server can be specified with

$$\langle enc_s((t_I, R, k), key(I)), t\rangle, \langle I, t'\rangle \dashv\!\lfloor\ 0 \leq t_S - t_I \leq 2 \wedge t \leq t_S \wedge t' \leq t_S\ \rfloor\!\mapsto \langle enc_s((t_S, I, k), key(R)), t_S\rangle \quad (3.3)$$

in which $key(U)$ represents the secret key shared between the server and the user U. Since the keys are only shared with the user and the server, We do not treat the key constructor as a public service. Besides, the names of the two participants should be known to the adversary, so we have services for publishing their names.

$$\dashv\!\lfloor\ \ \rfloor\!\mapsto \langle A[], t\rangle \quad (3.4)$$

$$\dashv\!\lfloor\ \ \rfloor\!\mapsto \langle B[], t\rangle \quad (3.5)$$

Event Services. In order to ensure the authenticity between participants, we introduce two special events *init* and *accept*. The *init* event is explicitly engaged by the initiator when a new protocol session starts, while the *accept* event is engaged by the protocol when the timed authentication is established successfully. According to [85], the timed authentication is correct if and only if every *accept* event is emitted with its corresponding *init* event engaged before, and the timing constraints should always be satisfied. For the WMF protocol, *Alice* engages an event *init* when she wants to start a session with R.

$$init(A[], R, [k], t_A), \langle R, t\rangle \dashv\!\lfloor\ t \leq t_A\ \rfloor\!\mapsto \langle enc_s((t_A, R, [k]), key(A[])), t_A\rangle \quad (3.6)$$

When the user *Bob* gets the message from the server, he decrypts it with his shared key $key(B[])$ and checks its freshness. If the timestamp checking is passed and the initiator is I, he then believes that he has established a timely authenticated connection under session key k with I and engages an *accept* event as follows.

$$\langle enc_s((t_S, I, k), key(B[])), t\rangle \dashv\!\lfloor\ t_B - t_S \leq 2\ \rfloor\!\mapsto accept(I, B[], k, t_B) \quad (3.7)$$

Additional Services. Introducing time allows to model systems which are not possible previously. For instance, some applications require that the passwords are used only if they are unexpired. One possible scenario is that the token $token(s, pw, t_k)$ can only be

opened within the lifetime $[t_k, t_k + d]$ of the password pw.

$$\langle token(s, pw, t_k), t_1\rangle, \langle pw, t_2\rangle \dashv\!\!\lfloor \begin{Bmatrix} t_1 \\ t_2 \end{Bmatrix} \leq \begin{Bmatrix} t_k + d \\ t \end{Bmatrix} \wedge t_k \leq t \rfloor\!\!\mapsto \langle s, t\rangle$$

If the adversary can obtain both of the token and the password within $[t_k, t_k + d]$, the secret s can be extracted from the token. Another possible service that could be accessible to the adversary is the brute force attack on the encrypted messages, which allows the adversary to extract the encrypted data without knowing the key. Suppose the least time of cracking the crypto is d, the attacking behavior can be modeled with

$$\langle Crypto(m, k), t\rangle \dashv\!\!\lfloor\, t' - t > d \,\rfloor\!\!\mapsto \langle m, t'\rangle.$$

For some ciphers like RC4 which is used by WEP, key compromise on a busy network can be conducted after a short time. Given an application scenario where such attack is possible and the attacking time has a lower bound d, we can model it as follows.

$$\langle RC4(m, k), t\rangle \dashv\!\!\lfloor\, t' - t > d \,\rfloor\!\!\mapsto \langle k, t'\rangle$$

Remarks. Even though the services specified in our framework can directly extract the message from the encryption without the key and so on, a given protocol can still guarantee correctness as long as proper timing checking is in place, e.g., authentication should be established before the adversary has the time to finish the brute-force attack.

3.2.3 Security Properties

In this work, we focus on verifying that the authentication between the two participants is timely, which means every $accept$ event is preceded by a corresponding $init$ event satisfying the timing constraints. Thus we formalize the *timed authentication* property by extending the definition in [85] as follows.

Definition 3.1. ***Timed Authentication.*** *In a timed security protocol, timed authentication holds for an* accept *event* e *with a set of* init *events* H *agreed on arguments encoded in the events and the timing constraints* B*, if and only if for every occurrence of* e*, all of the corresponding* init *events in* H *should be engaged before, and their timestamps should*

always satisfy the timing constraints B. *We denote the* timed authentication *query as* $e \leftarrow\!\!\lfloor\ B\ \rfloor\!\!\vdash H$. *In order to ensure general timed authentication, the arguments encoded in events should only be different variables and timestamps.*

We remark that the *timed authentication* defined above is non-injective. Because the legitimate run of WMF protocols requires that the authentication should be established within 4 time units, its query is modeled as follows.

$$accept(I, R, k, t) \leftarrow\!\!\lfloor\ t - t' \leq 4\ \rfloor\!\!\vdash init(I, R, k, t') \tag{3.8}$$

In Section 3.3, we present a verification algorithm to check the authentication. Since the verification for security protocol is generally undecidable [41], our algorithm cannot guarantee termination. Hence, we claim our attack searching algorithm as partial sound and partial complete under the condition of termination (partial correctness).

3.3 Verification Algorithm

Given the specification formalized in Section 3.2, our verification algorithm is divided into two phases. The attack searching service basis is constructed in the first phase so that attacks can be found in a straight forward method in the second phase. Specifically, every service consists of several inputs, one output and some timing constraints. When a service's input can be provided by another service's output, we could compose these two services together to form a composite service. In the first phase, our algorithm composes the services repeatedly until a fixed-point is reached. When such a fixed-point exists, we call it the *guided service basis*. However, the above process may not terminate because of two reasons. The first reason is the infinite knowledge deduction. For example, given two services $m \dashv\!\!\lfloor\!\rfloor\!\!\rightarrow h(m)$ and $h(m) \dashv\!\!\lfloor\!\rfloor\!\!\rightarrow h(h(m))$, we can compose them to obtain a new service $m \dashv\!\!\lfloor\!\rfloor\!\!\rightarrow h(h(m))$, which could be composed to the second service again. In this way, infinitely many composite services can be generated. The second reason is the infinite expansion of timing constraints. For instance, assume we have $S_0 = \langle enc(t', k), t_1\rangle \dashv\!\lfloor\ t'' - t' \leq 2 \wedge t_1 \leq t''\ \rfloor\!\rightarrow \langle enc(t'', k), t''\rangle$ and $S_1 = init(t, [k]) \dashv\!\lfloor\ t' - t \leq 2 \wedge t \leq t'\ \rfloor\!\rightarrow \langle enc(t', [k]), t'\rangle$ in the service basis. When we compose S_1 to S_0, their composition $S_2 =$

$init(t, [k])$ $-[\ t'' - t \leq 4 \wedge t \leq t''\ ]\!\!\rightarrow \langle enc(t'', [k]), t''\rangle$ has a larger range than S_1. Besides, we could compose S_2 to S_0 again to obtain an even larger range, so the service composition never ends. Since verification for untimed security protocol is undecidable, we, same as state-of-the-art tools like ProVerif, cannot handle the first scenario. We thus focus on solving the second scenario by approximating the timing constraints into a finite set. The fixed-point is then called the *approximated service basis*. When the over-approximation is applied, false alarms may be introduced into the verification result so that, generally, only partial completeness is preserved by our attack searching algorithm. Finally, we present our attack searching algorithm in the end of this section.

3.3.1 Service Basis Construction

In the first phase, our goal is to construct a set of services that allows us to find security attacks in the second phase. In order to construct such a service basis, new services are generated by composing existing services. In this way, the new composite services can also be treated as services directly accessible to the adversary and the algorithm continues until the fixed-point is reached, i.e., no new service can be generated. We use the most general unifier to unify the input and the output.

Definition 3.2. ***Most General Unifier.*** *If σ is a substitution for both messages m_1 and m_2 so that $\sigma m_1 = \sigma m_2$, we say m_1 and m_2 are unifiable and σ is an unifier for m_1 and m_2. If m_1 and m_2 are unifiable, the most general unifier for m_1 and m_2 is an unifier σ such that for all unifiers σ' of m_1 and m_2 there exists a substitution σ'' such that $\sigma' = \sigma''\sigma$.*

Since the adversary in our framework has the capability to generate new names and new timestamps, when a service input is a variable or a timestamp that is unrelated to other events in a service, the adversary should be able to generate a random event and use it to fulfill that input. In this way, that input can be removed in the composite service. Hence, we define service composition as follows. For simplicity, we define a *singleton* as a event of the form $\langle x, t\rangle$ where x is a variable or a timestamp.

Definition 3.3. ***Service Composition.*** *Let $S = [\ G\ ]\ H\ -[\ B\ ]\!\!\rightarrow e$ and $S' = [\ G'\ ]\ H'\ -[\ B'\ ]\!\!\rightarrow e'$ be two services. Assume there exists $e_0 \in H'$ such that e and e_0 are unifiable,*

their most general unifier is σ *and* $\sigma B \cap \sigma B' \neq \emptyset \wedge \sigma G \cap \sigma G' \neq \emptyset$. *The service composition of* S *with* S' *on a event* e_0 *is defined as*

$$S \circ_{e_0} S' = clear(\sigma([\ G \cap G'\ ]\ H \cup (H' - \{e_0\}))) \dashv[\ sim(\sigma B \cap \sigma B')\ ]\mapsto \sigma e'$$

where the function $clear$ *merges duplicated events from the inputs and removes any singleton* $\langle x, t\rangle$ *where* x *does not appear in other events of the rule, and the function* sim *removes timestamps that are no longer used in the composite service.*

When new composite services are added into the service basis, redundancies should be eliminated from the service basis. As the timing constraints can be viewed as a set of clock valuations which satisfy the constraints, they thus can be naturally applied with set operations, e.g., $B \subseteq B'$, $B \cap B'$, etc.

Definition 3.4. ***Service Implication.*** *Let* $S = [\ G\ ]\ H \dashv[\ B\ ]\mapsto e$ *and* $S' = [\ G'\ ]\ H' \dashv[\ B'\ ]\mapsto e'$ *be two services.* S *implies* S' *denoted as* $S \Rightarrow S'$ *if and only if* $\exists\sigma, \sigma e = e' \wedge G' \Rightarrow \sigma G \wedge \sigma H \subseteq H' \wedge B' \subseteq \sigma B$.

When services are composed in an unlimited way, infinitely many composite services could be generated. For instance, composing the symmetric encryption service (3.1) to itself on the event $\langle m, k\rangle$ leads to a new service encrypting the message twice, that is $\langle m, t\rangle, \langle k_1, t_1\rangle, \langle k_2, t_2\rangle \dashv[\ \ldots\ ]\mapsto \langle enc_s(enc_s(m, k_1), k_2), t'\rangle$, which can be composed to the encryption service again. In order to avoid these service compositions, we adopt a similar strategy proposed in [29] such that the unified event in the service composition should not be singletons. Moreover, the events in our system cannot be unified[2], thus we define $\mathbb{V}$ as a set of events that should not be unified, consisting of all events and singletons.

We denote $\beta(\alpha, \mathbb{R}_{init})$ as the fixed-point, where $\mathbb{R}_{init}$ is the initial service set and α is a service approximation function adopted during the construction. In order to compute $\beta(\alpha, \mathbb{R}_{init})$, we first define $\mathbb{R}_v$ based on the following rules, where $inputs(S)$ represents the inputs of service S.

1. $\forall S \in \mathbb{R}_{init}, \exists S' \in \mathbb{R}_v, S' \Rightarrow S$;

[2] *init* events only appear in the inputs and *accept* events only appear in the output.

2. $\forall S, S' \in \mathbb{R}_v, S \not\Rightarrow S'$;

3. $\forall S, S' \in \mathbb{R}_v$, if $\forall e_{in} \in inputs(S), e_{in} \in \mathbb{V}$ and $\exists f \notin \mathbb{V}, S \circ_e S'$ is defined, $\exists S'' \in \mathbb{R}_v, S'' \Rightarrow \alpha(S \circ_e S')$.

The first rule means that every initial service is implied by a service in $\mathbb{R}_v$. The second rule means that no duplicated service exists in $\mathbb{R}_v$. The third rule means that for any two services in $\mathbb{R}_v$, if the first service's inputs are in $\mathbb{V}$ and their composition exists, their approximated composition is also implied by a service in $\mathbb{R}_v$. These three rules means $\mathbb{R}_v$ is the minimal closure of the initial service set $\mathbb{R}_{init}$. Based on $\mathbb{R}_v$, we have

$$\beta(\alpha, \mathbb{R}_{init}) = \{S \mid S \in \mathbb{R}_v \wedge \forall e_{in} \in inputs(S) : e_{in} \in \mathbb{V}\}.$$

In the latter part of this section, α will be instantiated with no-approximation and over-approximation.

For any service, it is derivable from a service basis $\mathbb{R}$ if and only if there is a derivation tree that represents how the service is composed.

Definition 3.5. ***Derivation Tree.*** *Let $\mathbb{R}$ be a set of closed services and S be a closed service, where a closed service is a service with its output initiated by its inputs. Let S be a service in the form of $[\ G\ ]\ e_1, \ldots, e_n \dashv\!\!\lfloor\ B\ \rfloor\!\!\mapsto e$. S can be derived from $\mathbb{R}$ if and only if there exists a finite derivation tree defined as*

1. *edges in the tree are labeled by events;*

2. *nodes are labeled by the services in $\mathbb{R}$;*

3. *if a node labeled by S has incoming edges of $e_1^s, \ldots, e_n^s$, an outgoing edge of e^s, satisfying the untimed condition G^s and the timed condition B^s, then $S \Rightarrow [\ G^s\ ]\ e_1^s, \ldots, e_n^s \dashv\!\!\lfloor\ B^s\ \rfloor\!\!\mapsto e^s$;*

4. *the outgoing edge of the root is the event e;*

5. *the incoming edges of the leaves are $e_1, \ldots, e_n$.*

Additionally, (1) G is the intersection of all the untimed conditions in the derivation tree; (2) if all the timing constraints in the derivation tree form B, then the timing constraints for S is $sim(B)$, where sim removes timestamps that are no longer used. We name this tree as the derivation tree for S on $\mathbb{R}$.

Guided Service Basis. When no approximation is used in the service basis construction, the fixed-point is called *guided service basis* denoted as $\mathbb{R}_{guided} = \beta(\alpha_{guided}, \mathbb{R}_{init})$ where, for any service S, $\alpha_{guided}(S) = S$. In such a case, we prove that a service can be derived from the guided service basis whenever it can also be derived from the initial service set, and vice versa.

Theorem 3.6. *For any service S in the form of $[\ G\ ]\ H\ \text{-}[\ B\ ]\!\!\rightarrow e$ where $\forall e_{in} \in H : e_{in} \in \mathbb{V}$, S is derivable from $\mathbb{R}_{init}$ if and only if S is derivable from $\mathbb{R}_{guided}$.*

Before proving the above theorem, we prove a lemma first.

Lemma 3.7. *If $S_o \circ_e S'_o$ is defined, $S_t \Rightarrow S_o$ and $S'_t \Rightarrow S'_o$, then either there exists e' such that $S_t \circ_{e'} S'_t$ is defined and $S_t \circ_{e'} S'_t \Rightarrow S_o \circ_e S'_o$, or $S'_t \Rightarrow S_o \circ_e S'_o$.*

Proof. Let $S_o = [\ G_o\ ]\ H_o\ \text{-}[\ B_o\ ]\!\!\rightarrow e_o$, $S'_o = [\ G'_o\ ]\ H'_o\ \text{-}[\ B'_o\ ]\!\!\rightarrow e'_o$, $S_t = [\ G_t\ ]\ H_t\ \text{-}[\ B_t\ ]\!\!\rightarrow e_t$, $S'_t = [\ G'_t\ ]\ H'_t\ \text{-}[\ B'_t\ ]\!\!\rightarrow e'_t$. There should exist a substitution σ such that $\sigma e_t = e_o, \sigma H_t \subseteq H_o, \sigma e'_t = e'_o, \sigma H'_t \subseteq H'_o, G_o \Rightarrow \sigma G_t, G'_o \Rightarrow \sigma G'_t, \sigma B_t \supseteq B_o, \sigma B'_t \supseteq B'_o$. Assume $S_o \circ_e S'_o = clear(\sigma'([\ G_o \wedge G'_o\ ]\ H_o \cup (H'_o - e)))\ \text{-}[\ sim(\sigma' B_o \cap \sigma' B'_o)\ ]\!\!\rightarrow \sigma' e'_o$. We discuss the two cases as follows.

First case. Suppose $\exists e' \in H'_t$ such that $\sigma e' = e$. Since $S_o \circ_e S'_o$ is defined. e and e_o are unifiable. Let σ' be the most general unifier, $\sigma'\sigma e' = \sigma'\sigma e_t$, then e' and e_t are unifiable, therefore $S_t \circ_{e'} S'_t$ is defined. Let σ_t be the most general unifier, then $\exists \sigma'_t$ such that $\sigma'\sigma = \sigma'_t\sigma_t$. We have $S_t \circ_{e'} S'_t = clear(\sigma_t(H_t \cup (H'_t - e')))\ \text{-}[\ sim(\sigma_t B_t \cap \sigma_t B'_t)\ ]\!\!\rightarrow \sigma_t e'_t$. Since $\sigma'_t\sigma_t(H_t \cap (H'_t - e')) = \sigma'\sigma(H_t \cup (H'_t - e')) \subseteq \sigma'(H_o \cup (H'_o - e)), \sigma'_t\sigma_t e'_t = \sigma'\sigma e'_t = \sigma' e'_o$ and $\sigma'_t sim(\sigma_t B_t \cap \sigma_t B'_t) = sim(\sigma'_t\sigma_t B_t \cap \sigma'_t\sigma_t B'_t) = sim(\sigma'\sigma B_t \cap \sigma'\sigma B'_t) \supseteq sim(\sigma' B_o \cap \sigma' B'_o)$, and $\sigma'_t(\sigma_t G_t \wedge \sigma_t G'_t) = \sigma'_t\sigma_t G_t \wedge \sigma'_t\sigma_t G'_t = \sigma'\sigma G_t \cap \sigma'\sigma G'_t \Leftarrow \sigma' G_o \cap \sigma' G'_o$, we have $S_t \circ_{e'} S'_t \Rightarrow S_o \circ_e S'_o$.

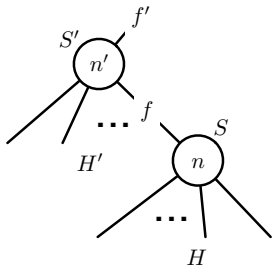


FIGURE 3.2: Two Nodes in Derivation Tree

Second case. Since $\forall e' \in H'_t$ such that $\sigma e' \neq e$, we have $\sigma H'_t \subseteq H'_o - e$. Thus $\sigma'\sigma H'_t \subseteq \sigma'(H_o \cup (H'_o - f))$, $\sigma'\sigma B'_t \supseteq \sigma' B'_o \supseteq \sigma' B_o \cap \sigma' B'_o$ $\sigma'\sigma G'_t \Leftarrow \sigma' G'_o \Leftarrow \sigma' G_o \cap \sigma' G'_o$, and $\sigma'\sigma e'_t = \sigma' e'_o$. Therefore $S'_t \Rightarrow S_o \circ_e S'_o$. □

Based on Lemma 3.7, we prove Theorem 3.6 as follows.

Proof. **(only if)** Assume S is derivable from $\mathbb{R}_{init}$, then there exists a derivation tree T_i for S on $\mathbb{R}_{init}$. Since a service S is removed from the basis only if it is implied by another service S', we have $\forall S \in \mathbb{R}_{init}, \exists S' \in \mathbb{R}_v, S' \Rightarrow S$.

As a result, we should be able to replace all the labels of nodes in T_i with services in $\mathbb{R}_v$ and get a new derivation tree T_v. Because some of the services are filtered out from $\mathbb{R}_v$ to $\mathbb{R}_{guided}$ when their input events do not all belong to $\mathbb{V}$, we further need to prove that the nodes in T_v can be composed together until a derivation tree T_{guided} is formed so that all the nodes in T_{guided} are labeled by services in $\mathbb{R}_{guided}$. To continue our proof, we assume that there exists two nodes n and n' in T_v and they are linked by an edge e as shown in Figure 3.2. We should have $S, S' \in \mathbb{R}_v$ such that $S \Rightarrow [\ G\ ]\ H \multimap [\ B\ ] \mapsto e$, $S' \Rightarrow [\ G'\ ]\ H' \multimap [\ \mathcal{B}'\ ] \mapsto e'$ and $e \in H'$. If $S_e = [\ G \cap G'\ ]\ (H \multimap [\ B\ ] \mapsto e) \circ_e (H' \multimap [\ \mathcal{B}'\ ] \mapsto e')$ is defined, according to Lemma 3.7, we could replace the two nodes with only one node in two different cases. In the first case, because $\mathbb{R}_v$ is the fixed-point of the service composition, there should exist $S'' \in \mathbb{R}_v$ such that $S'' \Rightarrow S_e$. In the second case, we can remove the node n and link its incoming links directly to the n', so that the new node n' is still implied by S'. We could continuously replace the nodes in the derivation tree until no node can be further processed and we denote the new tree as T_{guided}. For every node in

T_{guided}, we prove the services labeled to the nodes are in $\mathbb{R}_{guided}$ as follows.

- For the leaves of the tree, their incoming edges are labeled by the events in $\mathbb{V}$. So the leaves are labeled by services in $\mathbb{R}_{guided}$.
- For an inner node n' of the tree with all its children's service inputs in $\mathbb{V}$. Because n' cannot composed by its children, the inputs of the service labeled to n' should also be events in $\mathbb{V}$. So the services labeled to all the inner nodes are in $\mathbb{R}_{guided}$.

As a consequence, all the nodes in T_{guided} are labeled by services in $\mathbb{R}_{guided}$, so S is derivable from $\mathbb{R}_{guided}$.

(if) For every service in $\mathbb{R}_v$, it should be composed from existing services, which is in turn composed from the initial service set. Thus all the services in $\mathbb{R}_v$ should be derivable from $\mathbb{R}_{init}$. In the meanwhile, $\mathbb{R}_{guided}$ does not introduce extra services except for existing services in $\mathbb{R}_v$, so $\forall S \in \mathbb{R}_{guided}$, S is derivable from $\mathbb{R}_{init}$. □

Approximated Service Basis. New timestamps are often introduced in the service composition. When no longer used timestamps are removed from the composite service, the timing constraints can be deemed as extended for unification. On the other hand, given two services with the same inputs and output but they have different timing constraints, they may be indifferent if all of the different constraints have exceeded a ceiling. For instance, if the password has a fixed lifetime, its usefulness for the adversary remains the same when the password has already expired. Since these services can be deemed as the same, we remove their exceeded timing constraints to generalize their expressiveness. In this work, heuristically, we assume that every service is very likely to be used by the adversary for at least once in the attack trace and the timing constraints in the query also play important role in the reachability checking, so we set the ceiling as $1 + \sum max(B)$ in which B comes from the initial service set and the query. For instance, in the WMF protocol, the $max(B)$ is 2 for both of the service (3.3) and (3.7), 0 for other initial services, and 4 for the query, so we have the ceiling set as 9. We refer to the set of services with the ceiling U as approximated service basis $\mathbb{R}_{approx} = \beta(\alpha^U_{approx}, \mathbb{R}_{init})$. The service approximation function α^U_{approx} is defined as follows.

Definition 3.8. ***Service approximation with ceiling*** U. *Let* $S = [\ G\ ]\ H\ \dashv[\ B\ ]\!\!\rightarrow e$. *We define the service approximation with ceiling* U *as* $\alpha^{U}_{approx}(S) = [\ G\ ]\ H\ \dashv[\ B'\ ]\!\!\rightarrow e$. *For any two timestamps* t, t' *in the service* S, *if* $d(B, t, t') \leq U$, *then* $d(B', t, t') = d(B, t, t')$ *and* $c(B', t, t') = c(B, t, t')$; *else if* $d(B, t, t') > U$, *then* $d(B', t, t')$ *is* ∞ *and* $c(B', t, t')$ *is* $<$.

Since the timing constraints are enlarged after the approximation, false alarms may be introduced into verification result. However, according to the experiment results shown in Section 3.4, the false alarms could be prevented when the ceiling is properly configured. when the ceiling is properly configured. On the other hand, whenever a timed protocol is verified as correct under the approximation, it is guaranteed to be attack-free, which is the same as ProVerif.

Theorem 3.9. *Let* U *be the ceiling. For any service* S *in the form of* $[\ G\ ]\ H\ \dashv[\ B\ ]\!\!\rightarrow e$ *where* $\forall e_{in} \in H : e_{in} \in \mathbb{V}$, *if* S *is derivable from* $\mathbb{R}_{init}$, S *is also derivable from* $\mathbb{R}_{approx}$.

Proof. The proof for this theorem is almost the same as the *only if* proof for Theorem 3.6. Since service approximation only expands the timing constraints, given an initial service set $\mathbb{R}_{init}$ and a ceiling U, we should have that $\forall S \in \beta(\alpha_{guided}, \mathbb{R}_{init})$, $\exists S' \in \beta(\alpha^{U}_{approx}, \mathbb{R}_{init})$, $S' \Rightarrow S$. Because the only if for Theorem 3.6 is already proved, we should also have that S is derivable from $\mathbb{R}_{approx}$ whenever S is derivable from $\mathbb{R}_{init}$. □

3.3.2 Query Searching

When the query is violated by a service in the service basis, we call it a contradiction to the query. A service is a contradiction to the query if and only if its output event can be unified to the query's output, while it does not require all the predicate events in the query or it has a larger timing range than the query constraints. Thus, the contradiction is defined as follows.

Definition 3.10. ***Contradiction.*** *A service* $S = [\ G\ ]\ H\ \dashv[\ B\ ]\!\!\rightarrow e$ *is a contradiction to the query* $Q = e' \leftarrow\!\!\dashv[\ B'\ ]\vdash H'$ *if and only if* $G \neq false \wedge B \neq \emptyset$, e *and* e' *are unifiable with the most general unifier* σ *and* $\forall \sigma', \sigma'\sigma H' \not\subseteq \sigma H \vee \sigma B \not\subseteq \sigma'\sigma B'$.

If we rewrite the query Q into a service of $S_q = H' \dashv[\ B'\ ]\mapsto e'$, S is a contradiction to Q if and only if e' and e are unifiable with the most general unifier σ and we have $\sigma S_q \not\Rightarrow \sigma S$. According to Definition 3.1, events in the query only contain variables and timestamps that are different. Thus the *accept* event in S_q can be unified with any other *accept* event. The contradiction checking could then be simplified to check whether S outputs an *accept* event and satisfies $S_q \not\Rightarrow S$. Given the service basis $\mathbb{R}$, we thus search the attacks as follows.

$$\mathbb{R}_c = \{S | S \in \mathbb{R}, \text{the output of } S \text{ is an } \textit{accept} \text{ event} \wedge S_q \not\Rightarrow S\}$$

$\mathbb{R}_c$ consists of the contradiction instances. We prove its partial correctness as follows.

Theorem 3.11. ***Partial Soundness.*** *Assume $\mathbb{R}$ is $\mathbb{R}_{guided}$. Let Q be a query of $e' \leftarrow\!\!\dashv[\ B'\ ]\vdash H'$ and $S_q = H' \dashv[\ B'\ ]\mapsto e'$. There exists S derivable from $\mathbb{R}_{init}$ such that S is a contradiction to Q if there exists $S' \in \mathbb{R}$ such that the output of S' is an accept event and $S_q \not\Rightarrow S'$.*

Proof. **[$\mathbb{R} = \mathbb{R}_{guided}$]** Since $S' \in \mathbb{R}$ of which the output is an accept event and $S_q \not\Rightarrow S'$, according to the Definition 3.10, S' is a contradiction to the query Q. On the other hand, according to Theorem 3.6, since $S' \in \mathbb{R}$, S' is derivable from $\mathbb{R}_{init}$. □

Theorem 3.12. ***Partial Completeness.*** *Assume $\mathbb{R}$ is either $\mathbb{R}_{guided}$ or $\mathbb{R}_{approx}$. Let Q be a query of $e' \leftarrow\!\!\dashv[\ B'\ ]\vdash H'$ and $S_q = H' \dashv[\ B'\ ]\mapsto e'$. There exists S derivable from $\mathbb{R}_{init}$ such that S is a contradiction to Q only if there exists $S' \in \mathbb{R}$ such that the output of S' is an accept event and $S_q \not\Rightarrow S'$.*

Proof. **[$\mathbb{R} = \mathbb{R}_{guided}$ *or* $\mathbb{R}_{approx}$]** If S derivable from $\mathbb{R}_{init}$ is a contradiction to the Q, according to Theorem 3.6 (Theorem 3.9 resp.), there is a derivation tree T with its nodes labeled by services in $\mathbb{R}_{guided}$ ($\mathbb{R}_{approx}$ resp.) for S. Suppose the root of T is labeled by $S_r \in \mathbb{R}_{guided}$ ($S_r \in \mathbb{R}_{approx}$ resp.), the output of S_r is an accept event, and the inputs of S_r are events in $\mathbb{V}$. Because nodes in the derivation tree cannot be connected by events, the inputs corresponding to the edges connecting the children of root are singletons. If $S_q \Rightarrow S_r$, S_r implies the node and all of the input events in S_r is also in S, we have

$S_q \Rightarrow S$, which conflicts the precondition $S_q \not\Rightarrow S$. Thus $S_q \not\Rightarrow S_r$. As S_r is in $\mathbb{R}$ such that the output of S_r is an accept event, the theorem is then proved. □

Partial Soundness for Approximated Service Basis under Restriction. The partial soundness is not guaranteed for our verification algorithm when approximated service basis is used. However, when the initial services are specified in some restricted form, even though the approximated service basis is over-approximated, the partial soundness of our query searching algorithm can be proved as well. One possible restriction is that for any two timestamps t and t' in every initial service with B, $d(B', t, t')$ is required to be no less than 0. If the ceiling is set to be larger than $max(B_q) + 1$ where B_q is the timing constraints of the query, we prove the partial soundness of our verification algorithm as follows. First, we prove that, under this restriction, for any service S in the approximated service basis, we have a corresponding service S' in the guided service basis such that $S = \alpha^U_{approx}(S')$. Second, when the contradiction instance set $\mathbb{R}_e$ is not empty for the approximated service basis, we prove the existence of a corresponding attack instance in the guided service basis. According to the Theorem 3.6, the attack found in the guided service basis is guaranteed to be valid. So the protocol indeed has an attack and the partial soundness for the approximated service basis under the restriction is then proved.

Lemma 3.13. *Given an initial service set $\mathbb{R}_{init}$ and a ceiling U. Every service in $\mathbb{R}_{init}$ satisfies the restriction that for any two timestamps t and t' in the service with B, $d(B', t, t')$ is no less than 0. We have $\forall S \in \beta(\alpha^U_{approx}, \mathbb{R}_{init})$, $\exists S' \in \beta(\alpha_{guided}, \mathbb{R}_{init})$ such that $S = \alpha^U_{approx}(S')$.*

Proof. First, we need to prove the equation $\alpha^U_{approx}(\alpha^U_{approx}(S) \circ_e \alpha^U_{approx}(S')) = \alpha^U_{approx}(S \circ_e S')$ is hold when S and S' satisfy the restriction. Assume B is the timing constraint set for S, and B' for S', B_a for $\alpha^U_{approx}(S)$, B'_a for $\alpha^U_{approx}(S)'$ respectively. Given $d = d(B, t, t')$, $d' = d(B', t', t'')$, $d_a = d(B_a, t, t')$ and $d'_a = d(B'_a, t, t')$, we discuss different cases for d and d' as follows.

- If $d > U \wedge d' > U$, we have $d_a = \infty \wedge d'_a = \infty$, so $d + d' > U \wedge d_a + d'_a > U$.

- If $d > U \wedge 0 \leq d' \leq U$, we have $d_a = \infty \wedge d'_a = d'$, so $d + d' > U \wedge d_a + d'_a > U$.
- If $0 \leq d \leq U \wedge d' > U$, we have $d_a = d \wedge d'_a = \infty$, so $d + d' > U \wedge d_a + d'_a > U$.
- If $0 \leq d \leq U \wedge 0 \leq d' \leq U$, we have $d_a = d \wedge d'_a = d'$, so $d + d' = d_a + d'_a$.

Thus we have the timing constraints in $\alpha^U_{approx}(\alpha^U_{approx}(S) \circ_e \alpha^U_{approx}(S'))$ are the same as those in $\alpha^U_{approx}(S \circ_e S')$. Since the service approximation does not change the services inputs and output, the equation is thus valid. Given an approximated service in $\beta(\alpha^U_{approx}, \mathbb{R}_{init})$, there should exist a derivation tree labeled by services in $\mathbb{R}_{init}$ and the node is replaces by approximated services when two directly connected nodes are composed, according to the proof of Theorem 3.6. By repeatedly using the above equation during the node composition, we could delay the service approximation to a tree labeled by services in $\beta(\alpha_{guided}, \mathbb{R}_{init})$. Because the service approximation does not modify the service inputs and output, there should also be only one node in the tree. Hence we have $\forall S \in \beta(\alpha^U_{approx}, \mathbb{R}_{init}), \exists S' \in \beta(\alpha_{guided}, \mathbb{R}_{init})$ such that $S = \alpha^U_{approx}(S')$. □

Theorem 3.14. ***Partial Soundness under Restriction.*** *Assume $\mathbb{R}$ is $\mathbb{R}_{approx}$. Every service in $\mathbb{R}_{init}$ satisfies the restriction that for any two timestamps t and t' in the service with B, $d(B', t, t')$ is no less than 0. If the ceiling U is set to be larger than $max(B_q) + 1$ where B_q is the timing constraints of the query, $\mathbb{R} = \beta(\alpha^U_{approx}, \mathbb{R}_{init})$. Let Q be a query of $e' \leftarrow\!\!\lfloor\ B'\ \rfloor\!\!- H'$ and $S_q = H' -\!\lfloor\ B'\ \rfloor\!\!\rightarrow e'$. There exists S derivable from $\mathbb{R}_{init}$ such that S is a contradiction to Q if there exists $S' \in \mathbb{R}$ such that the output of S' is an accept event and $S_q \not\Rightarrow S'$.*

Proof. **[$\mathbb{R} = \mathbb{R}_{approx}$]** According to Lemma 3.13, there should exist a service S_g in $\beta(\alpha_{guided}, \mathbb{R}_{init})$ such that $S' = \alpha^U_{approx}(S_g)$. Since the ceiling U is set to be larger than $max(B_q)+1$, we have $S_q \not\Rightarrow S_g$ as well. Because of Theorem 3.11, we have S_g is derivable from $\mathbb{R}_{init}$ such that S_g is a contradiction to Q. The theorem is thus proved. □

Whether this restriction is applicable to the experiments evaluated is indicated in Section 3.4.

Remarks. Given a protocol with a valid attack, there should exist a derivation tree for that attack. Since we do not bound the number of events presented in a derivation tree (a composite service), we effectively deal with an unbounded number of sessions. The reason why our algorithm could work (i.e., terminate with correct result) is mainly because of two reasons. First, different from the explicit attack searching, we do not actively instantiate the variables in the services. So it becomes possible to represent the infinite adversary behaviors with a finite number of services. Second, we made a reasonable assumption in this work such that different nonces have different values. If the same nonce is generated in two sessions, those two sessions should be the same. Thus we merge them during the verification. As a consequence, even though we do not abstract the nonces used in the protocol as ProVerif does, this assumption could help us to find inconsistency in the service and remove the invalid ones from the service basis.

3.4 Evaluations

The flexibility and expressiveness of our service framework make it suitable for specifying and verifying timed security protocols, for instance, timed authentication protocols and distance bound protocols, etc. We have implemented our verifier TAuth in C++ with about 8K LoC. All the experiments shown in this section are conducted under Mac OS X 10.9.1 with 2.3 GHz Intel Core i5 and 16G 1333MHz DDR3. The TAuth verifier and the models shown in this section are available in [2].

We summarize some implementation choices in TAuth below. First, the timing constraints in the service are represented by Difference Bound Matrices (DBMs) [28]. Since timestamps are unified and new timestamps are introduced in the service composition, we use unique identifiers to distinguish the timestamps generated in the system so that different timestamps have different identifiers among services. Second, events in a service are merged when the encoded fresh nonces are evaluated to a same value. The reason is that the value of nonces generated in the session should be random, so different fresh nonces should have different values. For instance, if the session key k is initiated in the $init$ event, $init(A[], R_1, [k])$ and $init(I, R_2, [k])$ should be merged and the substitution $\{A[]/I, R_1/R_2\}$ should be applied to the service. If such events cannot be merged, the

Protocol	$\mathbb{R}_{guided}$			$\mathbb{R}_{approx}$			
	$\sharp\mathbb{R}$	Result	Time	$\sharp\mathbb{R}$	Result	Restriction	Time
Wide Mouthed Frog [35]	26	Attack [84]	3ms	26	Attack	SAT	4ms
Wide Mouthed Frog c [54]	19	Secure	3ms	19	Secure	SAT	3ms
Wide Mouthed Frog Lowe [84]	-	-	-	32	Secure	SAT	8ms
CCITT X.509(1) [40]	35	Attack [6]	4ms	35	Attack	SAT	3ms
CCITT X.509(1c) [6]	45	Secure	7ms	45	Secure	SAT	7ms
CCITT X.509(3) [40]	111	Attack [35]	52ms	111	Attack	SAT	51ms
CCITT X.509(3) BAN [35]	106	Secure	74ms	106	Secure	SAT	70ms
NS PK [94]	50	Attack [82]	6ms	50	Attack	SAT	6ms
NS PK Lowe [82]	51	Secure	8ms	51	Secure	SAT	9ms
NS PK Lowe Na Compromise [55]	51	Secure	8ms	51	Secure	SAT	8ms
NS PK Lowe Nb Compromise [55]	42	Attack [55]	3ms	42	Attack	SAT	3ms
NS PK Lowe NC Time [55]	48	Secure	10ms	48	Secure	UNSAT	10ms
SKEME [72]	77	Secure	73m	77	Secure	SAT	74ms
Auth Range [33, 38]	17	Secure	2ms	17	Secure	UNSAT	1ms
Ultrasound Dist Bound [103]	35	Attack [105]	2ms	35	Attack	UNSAT	2ms

Table 3.2: Verification results for timed authentication protocols

service is invalid. Third, we check the query contradiction on the fly when new services are composed. Whenever we find a contradiction, we stop the verification process and report the security flaw. This optimization can potentially give the early termination to the verification process when the protocol has security flaws.

Several different types of security protocols are analyzed in our experiments. In the experiments, *all* the protocols are proved or dis-proved in a short time as summarized in Table 3.2. For some protocols, the restriction mentioned in the Section 3.3.2 is applicable, so that the attack is guaranteed to be correct whenever it can be found, which is indicated in the table. Notice that, even though some protocols do not satisfy the restriction, all the attacks found in the experiments are valid. First, untimed protocols such as Needham-Schroeder series and SKEME are analyzed with TAuth. We use these protocols to show that TAuth can work with untimed protocols. Additionally, timed protocols like CCITT series are also checked by TAuth. However, the attacks found in these protocols are untimed. Furthermore, timed authentication protocols like the WMF series and the NS PK Lowe NC Time are correctly analyzed as well. We use these protocols to demonstrate that our approach can work with timed protocols and find timed attacks. Specifically, in the NS PK Lowe Nb Compromise version, the nonces generated by the responder in the protocol could be compromised [55], so the adversary could perform attacks to the proto-

Protocol	Result	TAuth	ProVerif	Scyther
NS PK	Attack	6ms	6ms	200ms
NS PK Lowe	Secure	8ms	5ms	177ms
NS PK Lowe Na Compromise	Secure	8ms	5ms	170ms
NS PK Lowe Nb Compromise	Attack	3ms	5ms	31ms

Table 3.3: Comparison with other untimed protocol verifiers.

col. Denning and Sacco [55] proposed a way to fix these security flaws by checking the timestamps. In the NS PK Lowe NC Time version, we assume that extra time is needed for the nonce compromise, so that freshness checking for the messages could ensure the authentication is attack-free. Notice that the service approximation only works for WMF Lowe version [84] in our experiment, that is a version of WMF fixed by Gavin Lowe, because it is the only protocol that cannot be early terminated by the on-the-fly algorithm (it is attack-free) and its timing constraints involve infinite expansion.

Moreover, we successfully analyze two distance bounding protocols, that are Auth Range [33, 38] and Ultrasound Dist Bound [103]. In the Auth Range protocol, the prover wants to convince the verifier that he is within a pre-agreed distance with the verifier. For instance, in a keyless entry system frequently adopted by cars, the prover is the remote key and verifier is the car. In the Auth Range protocol, it is assumed that the prover is honest and nothing can travel faster than light, so they could securely use the travel time of radio signals to measure the distance. In the Ultrasound Dist Bound protocol which has the same application scenario as the Auth Range protocol, the verifier uses radio signals to send requests while the prover uses ultrasound to return the answers. Since ultrasound travels much slower than radio and other processing time is negligible, the travel time of ultrasound dominates the whole protocol execution time. However, this protocol does not require the prover to be honest, so the prover can send his answer by either radio or ultrasound to others. When the adversary has a cooperator near the verifier, he can send the answer to the cooperator by radio and ask the cooperator to forward the answer by ultrasound to the verifier. As a consequence, the verifier can be convinced that the prover is within the distance even though the prover is not.

Finally, we compare our tool TAuth with other successful untimed protocol verifiers,

i.e., ProVerif [29] and Scyther [48]. The Needham Schroeder public key authentication protocols except for its timed variant are chosen for the comparison as timestamps are absent in these protocols. The comparison results are summarized in the Table 3.3. It can be seen that TAuth is almost as fast as ProVerif. TAuth is slightly slower mainly due to overhead on handling timing constraints. Thanks to the on-the-fly algorithm, TAuth is faster than ProVerif in finding the attack for the Lowe Nb Compromise version. Furthermore, TAuth is much faster than Scyther. Notice that Scyther could only verify the Lowe version and Lowe Na Compromise version with a bounded number of sessions while TAuth proves for infinitely many sessions.

3.5 Discussions

We present a service framework which can automatically verify the timed authentication protocols with an unbounded number of sessions. The partial correctness of our approach have been formally proved in this work. The experiment results for four different types of scenarios show that our framework is efficient and effective to verify a large range of timed security protocols. Even though we only check timed authentication properties for security protocols in this work, our framework could be easily extended to secrecy checking with timing constraints.

For future works, a throughout study on the termination of the algorithm would be very interesting. Since the problem of verifying security protocols is undecidable in general, we cannot guarantee the termination of our algorithm, but identifying the terminable scenario for practical security protocol could help the general adoption of our techniques. Our approach is inspired by the method used in ProVerif [29]. As is discussed in Section 3.3, TAuth is as terminable as ProVerif when the service approximation is used. However, the over-approximation also introduces false alarms. In order to remove the false alarms, as is discussed in Section 3.3, we can apply some restriction to the specification so that the found attacks are guaranteed to be valid. However, the restriction mentioned previously is quite restrictive because network latency, brute force attack, etc. cannot be specified under that restriction. Hence, how to restrict the specification in a practical way is another interesting future work direction.

Chapter 4

Parameterized Timed Security Protocol Verification

Quantitative timing is often explicitly used in systems for better security, e.g., the credentials for automatic website logon often has limited lifetime. Verifying timing relevant security protocols in these systems is very challenging as timing adds another dimension of complexity compared with the untimed protocol verification. In Chapter 3, we proposed an approach to check the correctness of the timed authentication in security protocols with fixed timing constraints. However, a more difficult question persists, i.e., given a particular protocol design, whether the protocol has security flaws in its design or it can be configured secure with proper parameter values? In this chapter, we answer this question by proposing a parameterized verification framework, where the quantitative parameters in the protocols can be intuitively specified as well as automatically analyzed. Given a security protocol, our verification algorithm either produces the secure constraints of the parameters, or constructs an attack that works for any parameter values. The correctness of our algorithm is formally proved. We implement our method into a tool called PTAuth and evaluate it with several security protocols. Using PTAuth, we have successfully found a timing attack in Kerberos V which is unreported before.

4.1 Introduction

Time could be a powerful tool in designing security protocols. For instance, distance bounding protocols rely heavily on time; session keys with limited lifetime are extensively used in practice to achieve better security. However, designing timed security protocols is more challenging than designing untimed ones because timing adds a range of attacking surface, e.g., the adversary might be able to extend the session key without proper authorization. Hence, it is important to have a formal verification framework to analyze the timed security protocols. In our work [77] illustrated in Chapter 3, we developed a verification algorithm to analyze whether a given protocol with fixed timing constraints is secure or not. In this work, we answer a more difficult question, i.e., given a security protocol with configurable parameters in the timing constraints, are there configuration methods which could guarantee security and what are they? Having an approach to answer the question is useful in a number of ways. Firstly, it can analyze, at once, all instances of the security protocols with different parameter values. Secondly, it allows the protocol designer to gain precise knowledge on the secure configuration of the parameters so as to choose the best values (e.g., in terms of minimizing the protocol execution time).

In general, parameterized timing constraints are necessary in various scenarios. First of all, they can be used to capture the general design of the protocols. For instance, since the lifetime of credentials are often related to the runtime information like network latency, it is best to keep them parameterized so that we can systematically find out their secure relations. Furthermore, parameterized timing constraints are necessary to model the properties of some special cryptographic primitives. For example, weak cryptographic functions, which are breakable by consuming extra time, may be used in the sensor networks for higher computing performance and lower power consumption. Since breaking different weak functions requires different attack time, in order to guarantee the correctness of the protocols in these sensor networks, we need to parameterize the attack time and

compute the secure configuration accordingly. Moreover, agencies often give suggestions on key crypto-period for cryptographic key management [22], so parameterized timing constraints can be used to model long term protocols.

Nevertheless, this is a highly non-trivial task. The challenges for designing timed protocol and providing proper parameter configuration are illustrated as follows. First, in the setting of timed authentication over the Internet, given the network is completely exposed to the adversary, we need to formally prove that the critical information cannot be leaked and the protocol works as intended under arbitrary attacking behaviors from the network. Second, timestamps are continuous values extracted from clocks to ensure the validity of messages and credentials. Analyzing the continuous timing constraints adds another dimension of complexity. Third, a protocol design might contain multiple timing parameters, e.g., the network latency and the session key lifetime, which could affect security of the system. Manually reasoning the least constrained and yet correct configuration for the parameters in complex protocols is extremely hard and error-prone. As a consequence, automatic analysis technique is needed for proving the correctness of the protocol and computing the parameter configurations.

Contributions. Our contributions in this work are summarized as follows. (1) We propose *timed logic rules* to specify parameterized timed protocols in Section 4.3 by extending our previous work [77] with parameterized timing constraint, secrecy query, etc. Additionally, we propose *timed applied π-calculus* to model the timed protocols in Section 4.4 and define its semantics based on the *timed logic rules* in Section 4.5. We thus facilitate intuitive specification method for timed security protocols with timing parameters. (2) Based on the *timed logic rules*, security protocols can be verified efficiently for an unbounded number of protocol sessions in our framework as shown in Section 4.6. Generally, in this work, we specify the adversary's capabilities in the security protocols as a set of Horn logic rules with parameterized timing constraints. Then, we compose these rules repeatedly until a

fixed-point is reached, so that we can check the desired security properties against them and compute the largest parameter configurations. The parameter configuration is represented by succinct constraints of the parameters. When the protocol could be secure with the right parameter values, our approach outputs a set of constraints on the parameters that are necessary for security. Otherwise, an attack is generated, which would work for any parameter configuration. We formally prove the correctness of our algorithm. (3) We implement our method as a tool named PTAuth. In order to handle the parameters in the timing constraints, we utilize the Parma Polyhedra Library (PPL) [17] in our tool to represent the relations between timestamps and parameters. We evaluate our approach with several security protocols in Section 4.7. During the experiment, we found a timing attack in the official document of Kerberos V [95] that has never been reported before.

4.2 Running Example: Wide Mouthed Frog

We use the same Wide Mouthed Frog (WMF) [35] protocol shown in Chapter 3 as a running example to illustrate how our approach works. WMF is designed for exchanging timely fresh session keys, ensuring that the key is generated by the protocol initiator within a short time when the protocol responder accepts it.

Syntax Hierarchy. Before describing the WMF protocol, we first introduce the syntax to represent the messages as shown in Table 4.1. *Messages* could be defined as *functions*, *names*, *nonces*, *variables* or *timestamps*. *Functions* can be applied to a sequence of *messages*; *names* are globally shared constants; *nonces* are freshly generated random values in sessions; *variables* are memory spaces for holding *messages*; and *timestamps* are clock readings extracted during the protocol execution. In addition, we introduce *parameters* to parameterize the timing constraints. The constraint function $\mathcal{C}(\mathbb{X})$ applies succinct constraints to $\mathbb{X}$, where $\mathbb{X}$ is a set of timestamps and parameters. Each succinct constraint can be written in a general form of $l(t_1, \ldots, t_n, \S p_1, \ldots, \S p_m) \sim 0$, where $\sim \in \{<, \leq\}$ and

Type	Expression	
Message(m)	$f(m_1, m_2, ..., m_n)$	(function)
	$a[]$	(name)
	$[n]$	(nonce)
	v	(variable)
	t	(timestamp)
Parameter(p)	$\S p$	(parameter)
Constraint(B)	$\mathcal{C}(t_1, t_2, \ldots, t_n$ $, \S p_1, \S p_2, \ldots, \S p_m)$	(timing relation)
Configuration(L)	$\mathcal{C}(\S p_1, \S p_2, \ldots, \S p_m)$	(parameter config)
Event (e)	$init(\star[d], m, t)$	(initialization)
	$join(\star m, \star t)$	(participation)
	$accept(\star[d], m, t)$	(acceptance)
	$know(\star m, t)$	(knowledge)
	$new(\star[n], loc[], m)$	(generation)
	$unique(\star u, \star loc[], m)$	(uniqueness)
	$leak(\star m)$	(leakage)
Rule(R)	$[\ G\ ]\ e_1, \ldots, e_n \multimap[\ B\ ]\rightarrowtail e$	(rule)

Table 4.1: Syntax Hierarchy Structure

l is a linear function. In the following, the symmetric encryption function is denoted as $enc_s(m, k)$, where m is the plaintext and k is the encryption key. Furthermore, all the messages transmitted in WMF is encrypted by the shared key represented as $sk(u)$, which is only known between the user u and the server. For simplicity, the concatenation function $tuple_n(m_1, m_2, \ldots, m_n)$ is written as $\langle m_1, m_2, \ldots, m_n \rangle$.

Events are constructed by attaching predicates to the message sequences. In our framework, we have seven different predicates:

- $init([d], m, t)$ means that a session with id $[d]$ has been initiated using the arguments in m at time t.
- other participants can engage $join(m, t)$ to show their participation in the protocol using the arguments in m at time t.
- similarly, the responder engages $accept([d], m, t)$ to indicate the protocol acceptance

under the arguments of m at time t in a session with id $[d]$;

- the knowledge event $know(m, t)$ means that the adversary knows the message m at the time t;

- the nonce generation event $new([n], loc[], m)$ stands for the generation of nonce $[n]$ at the location $loc[]$ where m records a message tuple that can be identified by the nonce $[n]$;

- the uniqueness event $unique(u, loc[], m)$ means that the message u is a unique value appeared at the location $loc[]$, where m records a message tuple that can be identified by $\langle u, loc[]\rangle$;

- the event $leak(m)$ is introduced to check the leakage of the secret message m that violates the secrecy property, as shown in the example later.

Comparing with the nonce generation event where the nonce $[n]$ should always be unique regardless of the location, the uniqueness checking of u is location dependent. For the same location, only one $unique$ event can have u; while for two different locations, two different $unique$ events can be claimed for the same u.

Generally, the timed logic rules represent the capabilities of the adversary, written as $[\ G\ ]\ e_1, e_2, \ldots, e_n\ \dashv[\ B\ ]\!\rightarrow\ e$, where G is a set of untimed guards, $\{e_1, e_2, \ldots, e_n\}$ is a set of premise events, B is a set of timing constraints and e is a conclusion event. It means that if the untimed guard condition G, the premise events $\{e_1, e_2, \ldots, e_n\}$ and the timing constraints B can be satisfied, the conclusion event is ready to occur. For simplicity, when G is empty, we omit the untimed guard condition in the rule. Notice some arguments of every event shown in Table 4.1 are marked with a special symbol $\star$. For every event, the $\star$ marked arguments forms the key of the event. When the events in the same rule have an identical key, they should be merged. For instance, given two *know*

events $know(m_1, t_1)$ and $know(m_2, t_2)$ in a rule's premises, if $m_1 = m_2$, we merge them by applying a substitution $\{t_2 \mapsto t_1\}$ to the whole rule.

Wide Mouthed Frog. The WMF protocol is a key exchange protocol consisting of three participants, i.e., the initiator *Alice*, the responder *Bob* and the server. It has the following steps.

$$
\begin{array}{ll}
\text{(1) } Alice \text{ engages} & : new([k], alice\_gen[], \langle A[], B[], t_A\rangle) \\
 & , init([k], \langle A[], B[], [k]\rangle, t_A) \\
\quad Alice \rightarrow Server & : \langle A[], enc_s(\langle t_A, B[], [k]\rangle, sk(A[]))\rangle \\
\text{(2) } Server \text{ checks} & : t_S - t_A \leq \S p_a \\
\quad Server \text{ engages} & : join(\langle A[], B[], [k]\rangle, t_S) \\
\quad Server \rightarrow Bob & : enc_s(\langle t_S, A[], [k]\rangle, sk(B[])) \\
\text{(3) } Bob \text{ checks} & : t_B - t_S \leq \S p_a \\
\quad Bob \text{ engages} & : new([b], bob\_gen[], \langle A[], B[], [k], t_S, t_B\rangle) \\
 & , accept([b], \langle A[], B[], [k]\rangle, t_B)
\end{array}
$$

First, *Alice* generates a fresh key $[k]$ at time t_A with the *new* event and engages an $init_A$ event to initiate the key exchange protocol with *Bob*. Second, *Alice* sends the fresh key with the current time t_A and *Bob*'s name to the server. Third, after receiving the request from *Alice*, the server checks the freshness of the timestamp t_A and accepts *Alice*'s request by engaging an $init_S$ event. Fourth, the server sends a new message to *Bob*, informing him that the server receives a request from *Alice* at time t_S to communicate with him using the key $[k]$. Fifth, *Bob* checks the timestamp and accepts the request from *Alice* if it is timely. The transmitted messages are encrypted under the users' shared keys.

Parameters. Whether or not WMF works relies on two crucial time parameters. The first parameter is the real network latency $\S p_d$ of the network, and the second one is the message delay $\S p_a$ allowed in the message freshness checking. $\S p_d$ is initially configured as $\S p_d > 0$ because the network latency should be positive. However, the exact value of $\S p_d$ depends on the network itself and thus cannot be fixed in the protocol design. Parameter $\S p_a$ on the other hand might be related to $\S p_d$'s value, which should be answered by the verification. That is to say, the values of the parameters are better modeled as unknown parameters

and we must be able to analyze the protocol without the concrete values of them. By introducing these two parameters, we want to make sure that the WMF protocol exchanges the secret session key successfully, and the correspondence between the request from Alice and the acceptance from Bob is timely. Hence, ideally a tool would automatically show us the secure configuration of $\S p_d$ and $\S p_a$. Because WMF has two message transmissions, we need to check whether $t_B - t_A \leq 2 * \S p_a$ is always satisfied.

4.3 Specifying Protocols using Timed Logic Rules

In this section, we introduce how to model the parameterized timed security protocols. Generally, protocols as well as their underlying cryptography foundation are represented by a set of Horn logic rule variants [29] as shown in Table 4.1. They, denoted as $\mathbb{R}_{init}$, represent the capabilities of the adversary in the protocol.

Adversary Model. We assume that an active attacker exists in the network, extending from the Dolev-Yao model [56]. The attacker can intercept all communications, compute new messages, generate new nonces and send any message he obtained. For computation, he can use all the publicly available functions, e.g., encryption, decryption, concatenation. He can also ask the genuine protocol participants to take part in the protocol based on his needs. Comparing our attack model with the Dolev-Yao model, attacking the weak cryptographic functions and compromising legitimate protocol participant are allowed by consuming extra time, as shown later in this section.

Rule Construction. Based on the adversary model described above, the interactions available to the adversary in the protocol can be represented by Horn logic rule variants guarded by timed checking conditions. Generally, every rule consists of a set of untimed guard conditions, several premise events, some timing constraints and one conclusion event as shown in Table 4.1. When the guard conditions, the premise events and the timing constraints in a rule are fulfilled, its conclusion event becomes available to the adversary. We

remove the brackets if the rule has no guard condition. For instance, since the symmetric encryption and decryption functions are publicly available in WMF, these capabilities of the adversary can be represented by the following two rules.

$$know(m, t_1), know(k, t_2) \dashv[\ t_1, t_2 \leq t\ ] \vdash know(enc_s(m, k), t) \tag{4.1}$$

$$know(enc_s(m, k), t_1), know(k, t_2) \dashv[\ t_1, t_2 \leq t\ ] \vdash know(m, t) \tag{4.2}$$

The rule (4.1) means that given a message m and a key k, the adversary can compute its encryption $enc_s(m, k)$, and the encryption can only be known after the message and the key are obtained. Similarly, the rule (4.2) shows the decryption capability of the adversary.

Furthermore, the adversary can register new accounts at the server, except for the existing ones of *Alice* and *Bob*. So, we have the following rule.

$$[c \neq A[] \wedge c \neq B[]]\ know(c, t_1) \dashv[\ t_1 \leq t\ ] \vdash know(sk(c), t) \tag{4.3}$$

For rules related to the protocol itself, they can be extracted from the protocol readily. For instance, the adversary can actively ask *Alice* to initiate the first step of the WMF protocol, so the messages in the second step can be intercepted from the network, which is shown by the rule (4.4). As Alice can initiate this protocol with any user at any time based on the adversary's needs, the constant $B[]$ is replaced with a variable R and $know(\langle R, t_A\rangle, t)$ is added to the premises of the rule, comparing with protocol description in Section 4.2.

$$know(R,t), new([k], alice\_gen[], \langle A[], R, t_A\rangle), init_A([k], \langle A[], R, [k]\rangle, t_A)$$
$$\dashv[\ t \leq t_A\ ] \vdash know(\langle A[], enc_s(\langle t_A, R, [k]\rangle, sk(A[]))\rangle, t_A) \tag{4.4}$$

Similarly, based on the server's behavior in the second step of WMF, we can construct the rule (4.5) shown below. Since the server provides its service to all of its users, *Alice* and *Bob*'s names are replaced by variables. The network latency and the message delay are captured by the parameterized constraints.

$$know(\langle I, enc_s(\langle t_I, R, k\rangle, sk(I))\rangle, t), join(\langle I, R, k\rangle, t_S)$$
$$\dashv[\ t_S - t \geq \S p_d \wedge t_S - t_I \leq \S p_a\ ] \vdash know(enc_s(\langle t_S, I, k\rangle, sk(R)), t_S) \tag{4.5}$$

Finally, *Bob* accepts the protocol when he receives the message from the server, indicating that the initiator is *Alice* and the request is fresh.

$$know(enc_s(\langle t_S, A[], k\rangle, sk(B[])), t), new([b], bob\_gen[], \langle A[], B[], k, t_S, t_B\rangle)$$
$$-[\ t_B - t \geq \S p_d \wedge t_B - t_S \leq \S p_a\ ]\mapsto accept([b], \langle A[], B[], k\rangle, t_B) \quad (4.6)$$

Additional Attack Rule. In addition to the attacker capabilities in the Dolev-Yao model, the attacker can compromise cryptographic primitives and legitimate protocol participants. For instance, we can model the brute-force attack on a weak encryption function. Given the name of the encryption function as $Crypto$ and the least time of cracking $Crypto$ as $\S d$, the attacking behavior can be modeled by the following rule.

$$know(Crypto(m, k), t_1) -[\ t - t_1 > \S d\ ]\mapsto know(m, t)$$

Additionally, for some ciphers, key compromise can be conducted under certain conditions. For example, for RC4 used by WEP, when a large number of ciphertexts are transmitted in the network, the encryption key can be compromised, which could be measured with time. Given an application scenario where such attack is possible and the attacking time has a lower bound $\S d$, we can model it as follows.

$$know(RC4(m, k), t_1)\rangle -[\ t - t_1 > \S d\ ]\mapsto know(k, t)$$

Authentication Query. Similar to our previous work [77], verifying the timely authentication is allowed in our framework. The timely authentication not only asks for the proper correspondence between the init and accept events but also requires the satisfaction of the timing constraints, formalized as follows. Extended from our previous work [78], given a timed security protocol, the timed non-injective authentication is satisfied if and only if for every acceptance of the protocol responder, the protocol initiator indeed initiates the protocol and the protocol partners indeed join in the protocol, agreeing on the protocol arguments and timing requirements. We formally define the non-injective timed authentication as follows.

Definition 4.1. ***Non-injective Timed Authentication.*** *In a timed protocol, non-injective timed authentication*

$$Q_n = accept \leftarrow\!\lfloor\ B\ \rfloor\!\vdash init, join_1, \ldots, join_n$$

holds if and only if for every occurrence of the event $accept$*, all of the corresponding events* $\{init, join_1, \ldots, join_n\}$ *are engaged before, and their timestamps should always satisfy the timing constraints* B*. In order to ensure the general timed authentication, the arguments encoded in the query events should only be variables and timestamps.*

The injective timed authentication additionally requires an injective correspondence between the protocol initialization and acceptance in addition to satisfaction of the non-injective timed authentication. Hence, the injective timed authentication, which ensures the infeasibility of the replay attack, is strictly stronger than the non-injective one.

Definition 4.2. ***Injective Timed Authentication.*** *The injective timed authentication, denoted as*

$$Q_i = accept \leftarrow\!\lfloor\ B\ \rfloor\!\rightarrowtail init, join_1, \ldots, join_n,$$

is satisfied by a timed protocol, if and only if (1) the non-injective timed authentication

$$Q_n = accept \leftarrow\!\lfloor\ B\ \rfloor\!\vdash init, join_1, \ldots, join_n,$$

is satisfied and (2) for any $init$ *event of* Q_i *occurred in the protocol, at most one* $accept$ *event with an unique id can occur, agreeing on the arguments in the events and the timing constraints in* B*.*

For simplicity, given a non-injective query $Q_n = accept \leftarrow\!\lfloor\ B\ \rfloor\!\vdash H$ and its injective version $Q_i = accept \leftarrow\!\lfloor\ B\ \rfloor\!\rightarrowtail H$, we have $inj(Q_n) = Q_i$ and $non\_inj(Q_i) = Q_n$. Similarly, given these two respective query sets $\mathbb{Q}_n$ and $\mathbb{Q}_i$, we have $inj(\mathbb{Q}_n) = \mathbb{Q}_i$ and $non\_inj(\mathbb{Q}_i) = \mathbb{Q}_n$.

In WMF, the authentication should be accepted by the responder R only if the request is made by the initiator I within $2 * \S p_a$. Thus, we have the following non-injective authentication query

$$Q_n^{WMF} = accept([b], \langle I, R, k\rangle, k_R) \leftarrow\!\lfloor k_R - k_S \leq \S p_a, k_S - k_I \leq \S p_a\ \rfloor\!\vdash$$
$$init([k], \langle I, R, k\rangle, t_I), join(\langle I, R, k\rangle, t_S) \quad (4.7)$$

and the corresponding injective authentication query $inj(Q_n^{WMF})$.

Secrecy Query. In this work, we extend the verification algorithm developed in our previous work [77] with secrecy checking that can be relevant to timing. Secrecy checking is introduced with additional rules that lead to the leak events, representing the leakage of the secret information.

Definition 4.3. ***Secrecy.*** *In a security protocol, secrecy holds for a message m if the event $leak(m)$ is unreachable when "$new_1, new_2, \ldots, new_n, know(m,t) \dashv\!\!\!\rightarrow leak(m)$" is added to $\mathbb{R}_{init}$, where $new_1, new_2, \ldots, new_n$ are the nonce generation events for all of nonces in m. Notice that different nonce generation events should have different locations so that they can be correctly identified in the protocol.*

For instance, according to the WMF protocol, a secret session key $[k]$ is sent over the network. In order to check the secrecy property of $[k]$, we add the following rule to $\mathbb{R}_{init}$ and then check the reachability of the leak event.

$$new([k], alice\_gen[], \langle A[], B[], t_A\rangle), know([k], t) \dashv\!\!\!\rightarrow leak([k]) \tag{4.8}$$

It means that if the session key $[k]$ generated by *Alice* for *Bob* can be known to the adversary, the secrecy property of the session key is invalid in WMF.

4.4 Specifying Protocols using Timed Applied π-calculus

In order to model the timed security protocols naturally, a high-level specification language should be provided. Hence, we develop *timed applied π-calculus* to specify timed security protocols, which extends the *applied π-calculus* [5] with time related operations and measurements. We use the Wide Mouthed Frog protocol [35] as a running example to demonstrate our modeling method.

Comparing with the syntax of the *applied π-calculus*, generating, checking and using timestamps are allowed in the *timed applied π-calculus*. The syntax of the *timed applied π-calculus* is shown in Table 4.2, which consists of five expression categories, i.e., *messages*, *parameters*, *timing constraints*, *parameter configurations* and *processes*.

Type	Expression	
Message(m)	$f(m_1, m_2, ..., m_n)$	(function)
	A, B, C	(name)
	n	(nonce)
	x, y, z	(variable)
	t	(timestamp)
Parameter(p)	p	(parameter)
Constraint(B)	$C(t_1, t_2, \ldots, t_n, p_1, p_2, \ldots, p_m)$	(timing constraint)
Configuration(L)	$C(p_1, p_2, \ldots, p_m)$	(parameter relation)
Process(P, Q)	0	(null process)
	$P \vert Q$	(parallel)
	$!P$	(replication)
	$\nu n.P$	(nonce generation)
	$\mu t.P$	(clock reading)
	if $m_1 = m_2$ then P [else Q]	(untimed condition)
	if B then P [else Q]	(timed condition)
	wait until $\mu t : B$ then P	(timing delay)
	let $m = f(\ldots)$ then P [else Q]	(function application)
	$c(m).P$	(channel input)
	$\overline{c}(m).P$	(channel output)
	check m as unique.P	(replay checking)
	$init(m)@t.P$	(initialization claim)
	$join(m)@t.P$	(participation claim)
	$accept(m)@t.P$	(acceptance claim)
	$secrecy(m).P$	(secrecy claim)

Table 4.2: Syntax of Timed Applied π-calculus

Generally, messages represent the data transmitted during the process execution. They can be hierarchal constructed by *functions*, *names*, *nonces*, *variables* and *timestamps*. *Functions* can be applied to a sequence of *messages*; *names* are globally shared constants; *nonces* are freshly generated random values in the processes; *variables* are memory spaces for holding *messages*; and *timestamps* are clock readings extracted during the process execution. Additionally, *parameters* stands for the timing settings that are generally agreed or globally exist in the protocol. By comparing them with the *timestamps* in the protocol, we can regulate the protocol execution trace.

Functions are defined as $f(m_1, m_2, \ldots, m_n) \Rightarrow m \ @ \ D$, where f is the function name, $m_1, m_2, \ldots, m_n$ are the input messages, m is the output message and D is the consumable timing range. For simplicity, we add some syntactic sugar as follows. (1) When $D =$

$[0, \infty)$ which is the largest timing range of functions, we omit '@ D' in the function definition. (2) When m is exactly the same as $f(m_1, m_2, \ldots, m_n)$, we similarly omit '$\Rightarrow m$' for short. For instance, the symmetric encryption function enc_s is originally defined as $enc_s(m, k) \Rightarrow enc_s(m, k)$ @ $[0, \infty)$. It means that a symmetric encryption $enc_s(m, k)$ can be generated from a message m and a key k using the function enc_s with no negative time. After the simplification, we can write its definition as $enc_s(m, k)$. Similarly, the symmetric decryption function dec_s can be defined as $dec_s(enc_s(m, k), k) \Rightarrow m$ with the same meaning of m and k. For illustration purpose, some frequently used functions are defined in Table 4.3.

The constraint set $B = C(t_1, t_2, \ldots, t_n, p_1, p_2, \ldots, p_m)$ represents a set of linear constraints over the timestamps and parameters, which can be used as checking condition and timing assumption in the protocol. For instance, given a parameter p_a as the maximum lifetime of messages in the protocol, the message receiver may check the message freshness with a timing constraint $t' - t \leq p_a$, when the message, generated by the sender at t, is received at t'. Meanwhile, given another parameter p_d as the minimum network delay, we may have $t' - t \geq p_d$ as a general constraint on the message transmission. Another constraint set $L = C(p_1, p_2, \ldots, p_m)$ stands for the global linear relations over the timing parameters. For example, the parameter relation $p_d \leq p_a$ should be generally satisfied because no message can be delivered otherwise. Before the verification, L will be configured to an initial relation. Later during the verification, whenever an attack is found, L will be updated with new constraints so as to remove the attack and preserve the security properties.

As shown in Table 4.2, processes are defined as follows. '0' is a null process that does nothing. '$P|Q$' is a parallel composition of processes P and Q. The replication '$!P$' stands for an infinite parallel composition of process P, which captures an unbounded number of protocol sessions running in parallel. The name restriction '$\nu n.P$' represents that a fresh nonce n is generated and bound to the process P. The time restriction '$\mu t.P$' similarly means that a timestamp t is read from the user's clock and bound to the process

Scheme	Definition	
Symmetric Encryption	$enc_s(m, k)$	(encryption)
	$dec_s(enc_s(m, k), k) \Rightarrow m$	(decryption)
Asymmetric Encryption	$pk(skey)$	(generate public key)
	$enc_a(m, pkey)$	(encryption)
	$dec_a(enc_s(m, pk(skey)), skey) \Rightarrow m$	(decryption)
Signature	$sign(m, skey)$	(generate signature)
	$check(sign(m, skey), pk(skey)) \Rightarrow true$	(check signature)
Hash	$hash(m)$	(generate hash value)
Tuple	$tuple_n(m_1, \ldots, m_n)$	(generate tuple)
	$\forall i \in \{1 \ldots n\}$: $get_i(tuple_n(m_1, \ldots, m_n)) \Rightarrow m_i$	(extract tuple)

Table 4.3: Cryptographic Function Definitions

P. The checking condition c in the expression 'if c then P [else Q]'[1] has two forms: (1) the untimed condition $m_1 = m_2$ is a symbolic equivalence checking between two messages; (2) the timed condition $C(t_1, t_2, \ldots, t_n, p_1, p_2, \ldots, p_m)$ is a numeric constraint over timestamps and parameters. When the condition c is valid, the process P is executed; otherwise, Q is executed. The timing delay expression 'wait until $\mu t : B$ then P' means that P is executed until the current clock reading satisfies the timing condition B. The function application expression 'let $m = f(m_1, \ldots, m_n)$ then P else Q' means if the function f is applicable to a sequence of messages $m_1, \ldots, m_n$, its result is bound to the message m in the process P; otherwise, the process Q is executed. The channel input expression '$c(m).P$' means that a message, bound to the name m, is received from the channel c before executing P. The channel output expression '$\overline{c}(m).P$' describes that the message m is sent to the channel c before executing the process P. The uniqueness checking expression 'check m as unique $.P$' ensures that the value of m has never been used before, comparing with other replications of this process. The uniqueness checking is particularly useful to prevent replay attacks in practice.

Additionally, four events are introduced in the process calculus to specify the security claims that can be made in the protocol.

- The protocol participant can engage a $secrecy(m)$ event to indicate that the message

[1] The expression in the brackets '$[E]$' means that E can be omitted.

m should be kept as a secret to the adversary.

- Right before the initiator finishes its role in starting the protocol, which is usually indicated by sending the last message, he emits an event $init(m)@t$, which means that a session has been initiated using the arguments in m at time t.

- When the responder finishes the protocol successfully, he engages an event $accept(m)@t$ to indicate the protocol acceptance under the arguments in m at time t.

- When other participants join the protocol run, they can engage an event $join(m)@t$ to show their participation in the protocol using the arguments in m at time t.

Overall, we use the $secrecy$ event to declare the secrecy property and use the $init$, $join$ and $accept$ events to check the authentication properties claimed by the protocol. As these events are closely related to the security properties, we explain them with the property definitions.

Notations. Several widely accepted notations for cryptographic protocol analysis are used in this chapter as follows. A variable m is bound to a process P when m is constructed by the function application expression 'let $m = f(m_1, \ldots)$ then P else Q' or the channel input expression '$c(m).P$' as shown in Table 4.2. When a variable m appears in a process P while it is not bound to P, it is a free variable in P. A process is *closed* when it does not have any free variable. $\sigma = \{x_1 \mapsto m_1, \ldots, x_n \mapsto m_n\}$ stands for the substitution that replaces the variables $x_1, \ldots, x_n$ with the messages $m_1, \ldots, m_n$ respectively. Given two messages m and m', when there exists a substitution σ such that $\sigma m = m'$, we say that m can be unified to m', denoted as $m \rightsquigarrow m'$; when no such substitution exists, we say that m cannot be unified to m', denoted as $m \not\rightsquigarrow m'$. Given two messages m_1 and m_2, if there exists a substitution σ such that $\sigma e_1 = \sigma e_2$, we say e_1 and e_2 are unifiable and σ is an unifier for e_1 and e_2. If e_1 and e_2 are unifiable, the most general unifier for e_1 and e_2 is an unifier σ such that for all unifiers σ' of e_1 and e_2 there exists a substitution σ'' such that $\sigma' = \sigma''\sigma$. The most general unifier for e_1 and e_2 is denoted as $mgu(m_1, m_2)$. For simplicity, the concatenation function $tuple_n(m_1, m_2, \ldots, m_n)$ is written as $\langle m_1, m_2, \ldots, m_n \rangle$ (or simply

$m_1, m_2, \ldots, m_n$ when no ambiguity is introduced). Given a tuple $x = \langle x_1, \ldots, x_i \rangle$ and a message y, their concatenation can be written as $x \cdot y = \langle x_1, \ldots, x_i, y \rangle$.

In the following, we again use the Wide Mouthed Frog (WMF) [35] protocol as an example to illustrate our specification language. For review purpose, WMF is designed to establish a timely fresh session key k from an initiator A to a responder B through a server S. In WMF, whenever a message is received, the receiver checks the message freshness before accepting it. To make a flexible specification, we thus use a parameter to represent the maximum message lifetime as p_a, ensuring that the message is received within p_a. Additionally, we consider another parameter p_d, which stands for the minimal network delay, during the verification. Since p_d is a timing parameter related to the network environment, it is not directly used in the protocol execution. Instead, it is a default and compulsory delay that applies to all of the network transmissions, so we add this delay to every channel inputs. In addition, we assume that the network latency is always positive, which makes the initial parameter configuration as $L_0 = \{p_d > 0\}$. Notice that a positive network delay is not compulsory in the protocol specification. However, setting the minimal network latency as $p_d \geq 0$ sometimes introduces a misleading result: the protocol can be verified as correct when p_d strictly equals to 0. Since the network latency p_d cannot be ensured as 0 in practice, the security protocol is thus proved as insecure instead. Because this final step of manual deduction is undesirable, we can remove it by simply requiring a positive network latency in the first place.

The Wide Mouthed Frog Protocol. The WMF protocol is a key exchange protocol that involves three participants, e.g., an initiator *Alice*, a responder *Bob* and a server S. *Alice* and *Bob* register their usernames as A and B at the server respectively. The generated key of a user u are written as $key(u)$. WMF then can be informally described as the following three steps.

(1) A generates a random session key k
$A \rightarrow S$: $\langle A, enc_s(\langle t_a, B, k\rangle, key(A))\rangle$
(2) S receives the request from A at t_s
S checks : $t_s - t_a \leq p_a$
$S \rightarrow B$: $enc_s(\langle t_s, A, k\rangle, key(B))$
(3) B receives the message from S at t_b
B checks : $t_b - t_s \leq p_a$
B accepts the session key k

First, A generates a fresh key k at time t_a and initiates the WMF protocol with B by sending the message $\langle A, enc_s(\langle t_a, B, k\rangle, key(A))\rangle$ to the server. Second, after receiving the request from A, the server checks the freshness of the timestamp t_a and accepts her request by forwarding a new message $enc_s(\langle t_s, A, k\rangle, key(B))$ to B, informing him that the server receives a request from A at time t_s to communicate with him using the key k. Third, B checks the message from the server as well and accepts the request from A if it is timely. All of the transmitted messages are encrypted under the users' long-term keys pre-registered at the server.

In order to verify WMF in a hostile environment, we assume that (1) the adversary can decide the protocol responder for A, (2) S provides its session key exchange service to all of its registered users and (3) the adversary can register as any user at the server, except for A and B. In WMF, because we are only interested in the protocol acceptance between the legitimate users, we ask B to only accept the requests from A. Additionally, a public channel c controlled by the adversary is used in this protocol for network communications.

Before the protocol starts, all of its participants need to register a secret long-term key at the server. We assume that A and B have already registered at the server using their names. Hence, the server can generate new keys for any other users (possibly personated by the adversary), which can be shown as the process P_r below.

$$P_r = c(u).\mathit{if}\ u \neq A \wedge u \neq B\ \mathit{then}\ \overline{c}(key(u)).0$$

In WMF, A takes a role of the initiator as specified by P_a below. She first starts the protocol by receiving an responder's name r from c, assuming that r is provided by the adversary. Then, A generates a session key k and claims k should be unknown to the adversary. Meanwhile, A emits an $init$ event, saying that A initializes the WMF proto-

col at time t_a, using the protocol arguments m_a. Notice that m_a is not clearly specified here, because we have not formally introduced the authentication property yet. The variable m_a is instantiated later in this section according to different types of authentication properties. Finally, the message $\langle A, enc_s(\langle t_a, r, k\rangle, sk(A))\rangle$ is sent from A to S. Since the initialization time t_a, the responder's namer r and the session key k are encrypted with A's long-term key, which is only known to A and the server, the adversary cannot obtain them directly.

$$P_a = c(r).\nu k.secrecy(k).\mu t_a.init(m_a)@t_a$$
$$.\overline{c}(\langle A, enc_s(\langle t_a, r, k\rangle, sk(A))\rangle).0$$

As specified by the process P_s, after the server receives a user's request y, it records the current time as t_s. It gets the initiator's name i from the unencrypted part of the request and use the key $key(i)$ to decrypt the encrypted part of the request. If the decryption function applies successfully, it stores the initialization time, the responder's name and the session key into t_i, r and k respectively. When the freshness checking $t_s - t_i \leq p_a$ is passed, the server then believes its participation in the current protocol run and engage a $join$ event at time t_s. Similar to m_a in the $init$ event, we specify the argument m_s with the authentication properties. Later, a new message encrypted by the responder's key, written as $enc_s(\langle t_s, i, k\rangle, key(r))$, is sent to the responder over the public channel.

$$P_s = c(y).\mu t_s.let\ i = get_1(y)\ then$$
$$let\ x = get_2(y)\ then\ let\ m = dec_s(x, key(i))\ then$$
$$let\ t_i = get_1(m)\ then\ let\ r = get_2(m)\ then$$
$$let\ k = get_3(m)\ then\ if\ t_s - t_i \leq p_a\ then$$
$$join(m_s)@t_s.\overline{c}(enc_s(\langle t_s, i, k\rangle, key(r))).0$$

Additionally, as shown in the process P_b, when B receives the request from the initiator through the server, B records his current time as t_b and tries to decrypt request as a tuple of the server's processing time t_s, the initiator's id i and the session key k. If $i = A$ and the freshness checking $t_b - t_s \leq p_a$ is valid, B then believes that the request is sent from

A within $2 * p_a$ (as the message freshness checking stacks) and engages the $accept$ event at time t_b.

$$P_b = c(x).\mu t_b.let\ m = dec_s(x, sk(B))\ then$$
$$let\ t_s = get_1(m)\ then\ let\ i = get_2(m)\ then$$
$$let\ k = get_3(m)\ then\ if\ i = A\ then$$
$$if\ t_b - t_s \leq p_a\ then\ accept(m_b)@t_b.0$$

Finally, we have a process P_p that broadcasts all of the public available names, e.g., *Alice* and *Bob*'s names.

$$P_p = \overline{c}(A).\overline{c}(B).0$$

The overall process P is an infinite parallel composition of the five processes described above.

$$P = (!P_r)|(!P_a)|(!P_s)|(!P_b)|(!P_p)$$

In this work, we discuss two types of security properties, i.e., authentication and secrecy. In order to clearly illustrate them, we introduce the formal adversary model adopted in this work first.

Adversary Model. We assume that an active attacker exists in the network, whose capability is defined based on and extended from the Dolev-Yao model [56]. The attacker can intercept all communications, compute new messages, generate new nonces and send any message he obtained. For computation, he can use all the publicly available functions, e.g., encryption, decryption, concatenation. He can also ask the genuine protocol participants to take part in the protocol based on his needs. Comparing our attack model with the Dolev-Yao model, attacking weak cryptographic functions and compromising legitimate protocol participants are allowed additionally. Notice that the adversary cannot emit the $accept$, $init$ and $secrecy$ events, because they can be only engaged by legitimate protocol users for checking properties. A formal definition of the adversary model in timed applied π-calculus is as follows.

Definition 4.4. ***Adversary Process.*** *The adversary process is defined as a closed timed applied π-calculus process S which does not emit the $init$, $join$ $accept$ and $secrecy$ events. Meanwhile, S can use all of the public functions.*

Timed Authentication. Typically, in the protocol, we have an initiator who starts the protocol and a responder who accepts the protocol. For instance, in WMF, *Alice* is the initiator and *Bob* is the responder. Additionally, other entities, who are called partners, can be involved during the protocol execution, such as the *server* in WMF. Given all of the protocol participants, the protocol authentication generally aims at establishing some common knowledge among them when the protocol successfully ends.

Since different participants take different roles in the protocol, we introduce the following three events for the initiator, the responder and the partners respectively. In these events, the argument m stands for the arguments used in the current protocol session.

- The protocol initiator emits $init$ event when he has initialized the protocol.
- The protocol responder emits $accept$ event when he has finished the protocol.
- The protocol partner emits $join$ event when his has participated in the protocol.

When any event is engaged, it means that the corresponding protocol participant believes his participation in a protocol run. Hence, the above events should be engaged immediately after the protocol participants successfully process all of the received messages according to their roles, as his knowledge of the protocol execution state cannot be increased after this point.

Based on the $init$, $join$ and $accept$ events, the protocol authentication then can be formally specified as the event correspondence. Extending from [77, 78], we discuss the non-injective and injective timed authentication properties in this work. Additionally, when different arguments are checked in event correspondence, they can be further categorized into agreement or synchronization properties.

Extended from our previous work [78], given a timed security protocol, the timed non-injective authentication is satisfied if and only if for every acceptance of the protocol responder, the protocol initiator indeed initiates the protocol and the protocol partners

indeed join in the protocol, agreeing on the protocol arguments and timing requirements. We formally define the non-injective timed authentication as follows.

Definition 4.5. ***Non-injective Timed Authentication.*** *The non-injective timed authentication, denoted as*

$$Q_n = accept \leftarrow\!\!\lfloor\ B\ \rfloor\!\!\vdash init, join_1, \ldots, join_n,$$

is satisfied by a closed process P, if and only if for any adversary process S, for every occurrence of an $accept$ event in $P|S$, the corresponding $init$ events and $join$ events in Q_n have occurred before in $P|S$, agreeing on the arguments in these events and the timing constraints in B.

The injective timed authentication additionally requires an injective correspondence between the protocol initialization and acceptance in addition to satisfaction of the non-injective timed authentication. Hence, the injective timed authentication, which ensures the infeasibility of the replay attack, is strictly stronger than the non-injective one.

Definition 4.6. ***Injective Timed Authentication.*** *The injective timed authentication, denoted as*

$$Q_i = accept \leftarrow\!\!\lfloor\ B\ \rfloor\!\!\rightarrowtail init, join_1, \ldots, join_n,$$

is satisfied by a closed process P, if and only if (1) the non-injective timed authentication

$$Q_n = accept \leftarrow\!\!\lfloor\ B\ \rfloor\!\!\vdash init, join_1, \ldots, join_n,$$

is satisfied by P and (2) for any adversary process S, corresponding to one $init$ event of Q_i occurred in $P|S$, at most one $accept$ event can occur in $P|S$, agreeing on the arguments in the events and the timing constraints in B.

(1) Timed Agreement Properties. When the message m encoded in the authentication events stands for the common knowledge established by the protocol among the participants, we call these timed authentication properties as timed agreement properties. The non-injective and injective timed agreement properties generally ensure the common knowledge establishment among the protocol participants under the timing restrictions.

Example 4.7. *In WMF, when B accepts the protocol, the common knowledge established among A, S and B should be the initiator's name, the responder's name and the session key. Hence, we specify the message m is different processes of WMF as follows.*

$$m_a = \langle A, r, k \rangle \qquad in\ P_a$$

$$m_s = \langle i, r, k \rangle \qquad in\ P_s$$

$$m_b = \langle i, B, k \rangle \qquad in\ P_b$$

The non-injective timed agreement then should be written as

$$\begin{aligned} Q_{na} = {} & accept(\langle i, r, k \rangle)@t_i \\ & \leftarrow\!\!\lfloor\ t_s - t_i \leq \S p_a \wedge t_r - t_s \leq \S p_a\ \rfloor\!\!\rightarrow \\ & \quad init(\langle i, r, k \rangle)@t_s, join(\langle i, r, k \rangle)@t_r \end{aligned} \tag{4.9}$$

where the responder accepts at time t_r, the server joins at time t_s and the initiator initializes at time t_i. Similarly, we have the injective timed agreement $Q_{ia} = inj(Q_{na})$.

(2) Timed Synchronization Properties. However, the above timed agreement properties do not necessarily guarantee the faithful message exchanges between protocol participants, so the messages received by the receiver may not be the same message sent by the sender in the protocol. Based on the synchronization defined in [52], when the message m encoded in the authentication events reflects the network input and output correspondence, we name these timed authentication properties after timed synchronization properties. The synchronization properties generally ensure that the messages exchanged in the protocol are untampered, so the message received by the receiver is the message sent from the sender for every network transmission.

Example 4.8. *In WMF, we first specify the arguments of the authentication events as follows to reflect the network communications.*

$$m_a = \langle r, \langle A, enc_s(\langle t_a, r, k \rangle, sk(A)) \rangle \rangle \qquad in\ P_a$$

$$m_s = \langle y, enc_s(\langle t_s, i, k \rangle, key(r)) \rangle \qquad in\ P_s$$

$$m_b = \langle x \rangle \qquad in\ P_b$$

Then, we specify the input and output correspondence in the non-injective timed synchronization property, written as follows.

$$Q_{ns} = accept(\langle s2b \rangle)@t_i$$
$$\leftarrow\!\!\lfloor\ t_s - t_i \leq \S p_a \wedge t_r - t_s \leq \S p_a\ \rfloor\!\!-$$
$$init(\langle r, a2s \rangle)@t_s, join(\langle a2s, s2b \rangle)@t_r$$

where the responder accepts at time t_r, the server joins at time t_s and the initiator initializes at time t_i. Notice that '$a2s$' is the message sent from A to S and '$s2b$' is the message sent from S to B. Similarly, we have the injective timed synchronization $Q_{is} = inj(Q_{ns})$.

Secrecy. When a protocol participant believes that a message m cannot be known to the adversary, he emits the event $secrecy(m)$ to claim for its secrecy.

Definition 4.9. ***Secrecy Property.*** *The secrecy property, denoted as $Q_s = secrecy(m)$ is satisfied by a closed process P, if and only if for any adversary process S, the message cannot be sent to the public channel c in $P|S$.*

4.5 Timed Applied π-calculus Semantics

The high-level *timed applied π-calculus* must facilitate efficient verification, e.g., with a concise and compact low-level semantics. We thus propose its semantics based on the Horn logic rules illustrated in Section 4.3. In this way, the timed security protocol thus can be naturally specified as well as efficiently verified. In this section, we define the semantics of the *timed applied π-calculus* illustrated previously based on the timed logic rules introduced in Section 4.2.

(1) Semantics of Functions. Since functions can be generally defined (before simplification) as

$$f(m_1, m_2, \ldots, m_n) = m@D,$$

their semantic rules can be accordingly written as

$$know(m_1, t_1), know(m_2, t_2), \ldots, know(m_n, t_n)$$
$$-\!\!\lfloor\ \forall i \in \{1 \ldots n\} : t - t_i \in D \wedge t' \geq t\ \rfloor\!\!\rightarrow know(m, t').$$

(2) Semantics of Processes. For processes, defining the semantics is more complex because we need to keep track of various protocol execution contexts. Thus, we introduce several semantic states. Generally, the semantically equivalent timed logic rules of a process P, can be denoted as $\lfloor P \rfloor TNUMGHB\sigma X$.

- T is a set of timestamps that are generated before executing P. We use it to order the timing of different behaviors. For instance, when a message is sent at time t in P, $\forall t' \in T : t' \leq t$ should be satisfied.

- N is a set of nonces generated before P. They can identify the current process until it terminates or sub-processes are forked.

- U is a set of value and location pairs that records the uniqueness checking happened before P. Whenever the uniqueness of u is checked at the location l, the pair $\langle u, l \rangle$ will be added to U.

- M is a tuple of messages that is determined by the current process's id. M consists of the process inputs, the generated timestamps and nonces. Their order in M follows the operation order in the process. The process outputs are not included in M because they can be determined by M as the process is deterministic.

- G is a set of untimed guard conditions that leads to the current process location. Given two messages m and m', two types of untimed guard conditions can be added to G, i.e., $m \neq m'$ and $m \not\rightsquigarrow m'$.

- B is a set of timing conditions that leads to the current process location, consisting of linear constraints over timestamps and parameters.

- σ is a naming substitution set that is applicable to P.

- X is a pair set of semi-completed timed logic rules and substitutions. Specifically, these rules do not have the new events for nonces in N and the $unique$ events for unique pairs in U, which can identify the current process. This is because the current process has not been finished yet. So we record the substitutions when the logic

rules are generated, and generate the completed rules when the process terminates or forks.

Given an initial process P_0, the Horn logic rules thus can be represented as $\lfloor P_0 \rfloor \emptyset\emptyset\emptyset\langle\rangle\emptyset\emptyset\emptyset\emptyset\emptyset$. In order to simplify the presentation of the translation, given a rule $R = [\ G\ ]\ H\ -[\ B\ ]\!\!\rightarrow e$, we write $R +_p H' = [\ G\ ]\ H \cup H' -[\ B\ ]\!\!\rightarrow e$.

First, we discuss three types of expressions that either terminate the current process or fork sub-processes, so that the nonces in N and the unique pairs in U cannot identify the current process after the expression. They are the null process '0', the process parallel '$P|Q$' and the process replication '$!P$'. Given the null process in $\lfloor 0 \rfloor TNUMGHB\sigma X$, the behavior tuple M of the current process is finally complete, so we can add the new events and the $unique$ events into the rules in X. The parallel composition process '$P|Q$' can be considered as three processes: P and Q are executed without being identified by the nonces in N and the pairs in U, and the current process terminates with the session ids from N. The infinite process replication '$!P$' can be similarly described as an infinite process parallel $P|\ldots|P$. Upon constructing the new event for a nonce $[n]$ in N, we introduce a function $loc([n])$ to provide an unique name depending on the generation location of $[n]$. In practice, the function loc can be implemented as $\{l = l + 1; return\ l;\}$ where l is a global variable initialized as 0.

$$\begin{aligned}
\text{Given } H' = \{&new([n], loc([n]), M) \mid [n] \in N \wedge M \neq \langle\rangle\} \\
&\cup \{update(u, l[], M) \mid \langle u, l[]\rangle \in U \wedge M \neq \langle\rangle\} \\
\lfloor 0 \rfloor TNUMGHB\sigma X &= \{\sigma'(R +_p H') \mid \langle R, \sigma'\rangle \in X\} \\
\lfloor P|Q \rfloor TNUMGHB\sigma X &= \lfloor 0 \rfloor TNUMGHB\sigma X \\
&\cup \lfloor P \rfloor T(\emptyset)(\emptyset)MG(H \cup H')B\sigma(\emptyset) \\
&\cup \lfloor Q \rfloor T(\emptyset)(\emptyset)MG(H \cup H')B\sigma(\emptyset) \\
\lfloor !P \rfloor TNUMGHB\sigma X &= \lfloor 0 \rfloor TNUMGHB\sigma X \\
&\cup \lfloor P \rfloor T(\emptyset)(\emptyset)MG(H \cup H')B\sigma(\emptyset)
\end{aligned}$$

Second, for the nonce and timestamp generation expressions, we add the nonces and

the timestamps into the nonce set N and the timestamp set T respectively. Furthermore, for the timestamp generation, we also add timing constraints to describe that the newly generated timestamp is larger or equal to the previously generated ones.

$$\lfloor \nu n.P \rfloor TNUMGHB\sigma X$$
$$= \lfloor P \rfloor T(N \cup \{[n]\})U(M \cdot [n])GHB\sigma X$$
$$\lfloor \mu t.P \rfloor TNUMGHB\sigma X$$
$$= \lfloor P \rfloor (T \cup \{t\})NU(M \cdot t)GH(B \cap \{t' \leq t \mid t' \in T\})\sigma X$$

Third, four conditional expressions exist in timed applied π-calculus. The inequivalence condition between messages should be included in G, while the timing constraints should be added to B. The timing delay expression requires that the current timing satisfies some timing constraint.

$\lfloor$if $m_1 = m_2$ then P else $Q\rfloor TNUMGHB\sigma X$

$= \lfloor P \rfloor TNUMGHB(mgu(m, m') \cdot \sigma)X \cup \lfloor P \rfloor TNUM(G \wedge m_1 \neq m_2)HB\sigma X$

$\lfloor$if B_0 then P else $Q\rfloor TNUMGHB\sigma X$

$= \lfloor P \rfloor TNUMGH(B \cap B_0)\sigma X \cup (\cup\{\lfloor Q \rfloor TNUMGH(B \cap \neg c)\sigma X \mid \forall c \in B_0\})$

$\lfloor$wait until $\mu t : B_t$ then $P\rfloor TNUMGHB\sigma X$

$= \lfloor P \rfloor (T \cup \{t\})NU(M \cdot t)GH(B \cap \{t' \leq t \mid t' \in T\} \cap B_t)\sigma X$

Given function f defined as $f(m'_1, \ldots, m'_n) \Rightarrow m'@D$

and $\sigma' = mgu(\langle m_1, \ldots, m_n \rangle, \langle m'_1, \ldots, m'_n \rangle)$, we have

$\lfloor$let $m = f(m_1, \ldots, m_n)$ then P else $Q\rfloor TNUMGHB\sigma X$

$= \lfloor \mu t_1$.wait until $\mu t_2 : t_2 - t_1 \in D$ then $P \rfloor TNUMGHB(\{m \mapsto m'\} \cdot \sigma' \cdot \sigma)X$

$\cup \lfloor Q \rfloor TNUM(G \wedge \langle m'_1, \ldots, m'_n \rangle \not\rightsquigarrow \langle m_1, \ldots, m_n \rangle)HB\sigma X$

Fourth, network communications can happen in the timed applied π-calculus. For network input, we record the timing when it is received and add a premise event as a requirement to know that message. For every network output, we store an incomplete

rules with the current substitution into X, considering that the output can be sent to the network and known to the adversary as a result when all the premise events, untimed guards and timing constraints are satisfied.

Given t as a new timestamp, we have

$$\lfloor c(m).P \rfloor TNUMGHB\sigma X$$
$$= \lfloor P \rfloor (T \cup \{t\}) NU(M \cdot m) G(H \cup \{know(m,t)\})(B \cup \{t' \leq t \mid t' \in T\})\sigma X$$
$$\lfloor \bar{c}(m).P \rfloor TNUMGHB\sigma X$$
$$= \lfloor P \rfloor TNUMGHB\sigma(X \cup \{\langle [G]\ H \multimap [\ B \cap \{t - t' \geq \S p_d \mid t' \in T\}\ ] \mapsto know(m,t), \sigma \rangle\})$$

Fifth, we can check the uniqueness of values in the process, which could be particularly useful to prevent replay attacks, ensuring the injective timed authentication. In practice, the uniqueness checking is usually implemented by maintaining a database and comparing the new values with the existing ones. Notice that we reuse the function $loc(m)$ to calculate the checking location.

$$\lfloor \text{check } m \text{ as unique}.P \rfloor TNUMGHB\sigma X$$
$$= \lfloor P \rfloor TN(U \cdot \langle m, loc(m) \rangle) MGHB\sigma X$$

Sixth, three types of authentication events can be engaged in the process. The $join$ event is similar to the $join$ expression in the calculus. However, for the $init$ and $accept$ events, although their meanings are preserved in the timed logic rules, in order to check the injective authentication property, we add an additional argument to them to represent the session id. In fact, any nonce generated in the current session that is stored in N can be used as the session id. When N is an empty set, we active generate a nonce before engaging these two events. Both of the $init$ and $join$ events are added into the rule premises,

while the $accept$ event acts as the rule conclusion.

$$\lfloor init(m)@t.P \rfloor TNUMGHB\sigma X$$

$$= \text{ if } \nexists [d] \in N \text{ then } \lfloor \nu n.init(m)@t.P \rfloor TNUMGHB\sigma X$$

$$\text{else } \lfloor P \rfloor TNUMG(H \cup \{init([d], m, t)\})(B \cup \{t \geq t' | \forall t' \in T\})\sigma X$$

$$\lfloor join(m)@t.P \rfloor TNUMGHB\sigma X$$

$$= \lfloor P \rfloor TNUMG(H \cup \{join(m, t)\})(B \cap \{t \geq t' | \forall t' \in T\})\sigma X$$

$$\lfloor accept(m)@t.P \rfloor TNUMGHB\sigma X$$

$$= \text{ if } \nexists [d] \in N \text{ then } \lfloor \nu n.accept(m)@t.P \rfloor TNUMGHB\sigma X$$

$$\text{else } \lfloor P \rfloor TNUMGHB\sigma(X \cup \{\langle [G]\ H \dashv [\ (B \cap \{t \geq t' | \forall t' \in T\})\ ] \mapsto accept([d], m, t), \sigma \rangle\})$$

Seventh, the last expression in the timed applied π-calculus is the secrecy claim. However, the secrecy property is checked as an absence of information leakage as shown in Section 4.6, so we use the event $leak(m)$ as an contradiction event against $secrecy(m)$.

$$\lfloor secrecy(m).P \rfloor TNUMGHB\sigma X$$

$$= \lfloor P \rfloor TNUMGHB\sigma(X \cup \{\langle [G]\ H \cup \{know(m, t)\} \dashv [\ B\ ] \mapsto leak(m), \sigma \rangle\})$$

4.6 Verification Algorithm

Given a rule $R = [\ G\ ]\ H \dashv [\ B\ ] \mapsto e$, a set of rules $\mathbb{R}$ and a parameter configuration L, we use $\alpha(R, L) = [\ G\ ]\ H \dashv [\ B \cap L\ ] \mapsto e$ and $\alpha(\mathbb{R}, L) = \{\alpha(R) | R \in \mathbb{R}\}$ to represent the rules under the configuration L. Since the initial rules $\mathbb{R}_{init}$ can be extracted from the protocol as shown in Section 4.3, the satisfaction of an authentication query Q then depends on whether the adversary can actively guide the protocol to reach the *accept* event based on $\alpha(\mathbb{R}_{init}, L)$ without engaging the corresponding *init* events in Q or satisfying the timing constraints. Similarly, the verification of the secrecy query needs to check that the *leak* event is unreachable based on $\alpha(\mathbb{R}_{init}, L)$. In this section, we focus on computing the largest parameter configuration that ensures the correctness of the desired authentication and secrecy properties.

Given any parameter configuration L, in order to determine whether a query Q is satisfied by $\alpha(\mathbb{R}_{init}, L)$, we can adapt the verification algorithm in [77]. However, there might be infinitely many possible parameter configurations. Thus, in this work, we develop an approach to handle the parameters symbolically. Specifically, the verification is divided into two sequential phases: the rule basis construction phase and the query searching phase. In the rule base construction phase, we generate new rules by composing two rules (through unifying the conclusion of the first rule and the premise of the second rule). Our verification algorithm uses this method repeatedly to generate new rules until a fixed-point is reached. This fixed-point is called the *rule basis* if it exists. Subsequently, in the query searching phase, the query is checked against the *rule basis* to find counter examples. Generally, we need to check the event correspondence as well as the parameterized timing constraints, the verification either proves the correctness of the protocol by providing the secure configuration of the parameters (represented as succinct constraints), or reports attacks because no parameter configuration can be found. Since the verification for security protocol is generally undecidable [41], our algorithm cannot guarantee termination. However, as shown in Section 4.7, our algorithm can terminate on most of the evaluated security protocols. Additionally, limiting the number of protocol sessions is allowed in our framework which would guarantee the termination of our algorithm.

Rule Basis Construction. Before constructing the rule basis, we need to review some basic concepts introduced in Chapter 3 first:

- If σ is a substitution for both events e_1 and e_2 such that $\sigma e_1 = \sigma e_2$, we say e_1 and e_2 are unifiable and σ is an unifier for e_1 and e_2. If e_1 and e_2 are unifiable, the most general unifier for e_1 and e_2 is an unifier σ such that for all unifiers σ' of e_1 and e_2 there exists a substitution σ'' such that $\sigma' = \sigma''\sigma$.
- Given two rules $R = [\ G\ ]\ H\ -[\ B\ ]\!\!\rightarrow e$ and $R' = [\ G'\ ]\ H'\ -[\ B'\ ]\!\!\rightarrow e'$, if e and $e_0 \in H'$ can be unified with the most general unifier σ such that $\sigma G \wedge \sigma G'$ can be valid, their composition is denoted as $R \circ_{e_0} R' = \sigma([\ G \wedge G'\ ]\ H \cup (H' - \{e_0\}))\ -[\ \sigma(B \cap B')\ ]\!\!\rightarrow \sigma e'$.

- Additionally, given the above two rules R and R', we define R implies R' denoted as $R \Rightarrow R'$ when $\exists \sigma, \sigma e = e' \wedge G' \Rightarrow \sigma G \wedge \sigma H \subseteq H' \wedge B' \subseteq \sigma B$.

We construct the rule basis $\beta(\mathbb{R}_{init})$ based on the initial rules $\mathbb{R}_{init}$. Firstly, we define $\mathbb{R}_v$ as follows, representing the minimal closure of the initial rules $\mathbb{R}_{init}$. (1) $\forall R \in \mathbb{R}_{init}, \exists R' \in \mathbb{R}_v, R' \Rightarrow R$, which means that every initial rule is implied by a rule in $\mathbb{R}_v$. (2) $\forall R, R' \in \mathbb{R}_v, R \not\Rightarrow R'$, which means that no duplicated rule exists in $\mathbb{R}_v$. (3) $\forall R, R' \in \mathbb{R}_v$ and $R = [G]\ H\ -[\ B\ ]\rightarrow\ e$, if $\forall e' \in H, e' \in \mathbb{V}$ and $\exists e_0 \notin \mathbb{V}, S \circ_{e_0} S'$ is defined, then $\exists S'' \in \mathbb{R}_v, S'' \Rightarrow R \circ_{e_0} R'$, where $\mathbb{V}$ is a set of events that can be provided by the adversary. In this work, $\mathbb{V}$ consists of the *init* events, the *join* events, the *new* events, the *unique* events and the $know(x, t)$ event where x is a variable or a timestamp. The third rule means that for any two rules in $\mathbb{R}_v$, if all premises of one rule are trivially satisfiable and their composition exists, their composition is implied by a rule in $\mathbb{R}_v$. Based on $\mathbb{R}_v$, we can calculate the rule basis as follows.

$$\beta(\mathbb{R}_{init}) = \{R \mid R = [G]\ H\ -[\ B\ ]\rightarrow\ e \in \mathbb{R}_v \wedge \forall e' \in H : e' \in \mathbb{V}\}$$

Theorem 4.10 means that the rules in $\alpha(\mathbb{R}_{init}, L)$ is equivalent to those in $\alpha(\beta(\mathbb{R}_{init}), L)$. Since the premises of the rules in $\alpha(\beta(\mathbb{R}_{init}), L)$ are trivially satisfiable according to the function β, the attack searching based on $\alpha(\beta(\mathbb{R}_{init}), L)$ would be much easier.

Theorem 4.10. *For any rule R in the form of $[G]\ H\ -[\ B\ ]\rightarrow\ e$ where $\forall e' \in H : e' \in \mathbb{V}$, R is derivable from $\alpha(\mathbb{R}_{init}, L)$ if and only if R is derivable from $\alpha(\beta(\mathbb{R}_{init}), L)$.*

Proof. Given a derivation tree T of R, we define $\Gamma(T, L)$ as a derivation tree where every node's label R' is replaced with $\alpha(R', L)$. According to Theorem 3.6, $R = [\ G\ ]\ H\ -[\ B\ ]\rightarrow\ e$ is derivable from $\mathbb{R}_{init}$ if and only if R is derivable from $\beta(\mathbb{R}_{init})$. It means that we can construct a derivation tree T of R based on $\mathbb{R}_{init}$ if and only if we can construct a derivation tree T' of R based on $\beta(\mathbb{R}_{init})$. After applying the configuration L to all of the labels of T, we have the following two conditions.

- If $B \cap L \neq \emptyset$, $\Gamma(T, L)$ becomes a derivation tree of $\alpha(R, L)$ based on $\alpha(\mathbb{R}_{init}, L)$, and $\Gamma(T', L)$ becomes a derivation tree of $\alpha(R, L)$ based on $\alpha(\beta(\mathbb{R}_{init}), L)$.

- If $B \cap L = \emptyset$, $\alpha(R, L)$ becomes invalid, so both of $\Gamma(T, L)$ and $\Gamma(T', L)$ do not exist.

Hence, $\alpha(R, L)$ is derivable from $\alpha(\mathbb{R}_{init}, L)$ if and only if $\alpha(R, L)$ is derivable from $\alpha(\beta(\mathbb{R}_{init}), L)$. The theorem is then proved. □

Query Searching. A rule is a contradiction rule to the non-injective authentication query if and only if its conclusion event is an *accept* event, while it does not require all the *init* and *join* events as premises or it has looser timing constraints comparing with those in the query. Otherwise, it is an obedience rule to the non-injective agreement query.

Definition 4.11. ***Non-injective Authentication Contradiction and Obedience.*** *A rule* $R = [\ G\ ]\ H\ \dashv[\ B\ ]\mapsto\ e$ *is a contradiction to the non-injective authentication query* $Q_n = accept \leftarrow[\ B'\ ]\vdash\ H'$ *denoted as* $Q_n \nvdash R$ *if and only if* $G \neq false \wedge B \neq \emptyset$, e *and* $accept$ *are unifiable with the most general unifier* σ *such that* $\forall e' \in H, e' \in \mathbb{V}$ *and* $\forall \sigma', (\sigma'\sigma H' \not\subseteq \sigma H) \vee (\sigma B \not\subseteq \sigma'\sigma B')$. *On the other hand, it is an obedience to* Q_n *denoted as* $Q_n \vdash R$ *if and only if* $G \neq false \wedge B \neq \emptyset$, e *and* $accept$ *are unifiable with the most general unifier* σ *such that* $\forall e' \in H, e' \in \mathbb{V}$ *and* $\exists \sigma', (\sigma'\sigma H' \subseteq \sigma H) \wedge (\sigma B \subseteq \sigma'\sigma B')$.

Furthermore, the injective authentication query is violated if and only if (1) there exists a contradiction to the non-injective version of the query, or (2) given two obedience rules to the non-injective version of the query, when the corresponding *init* events have identical session ids, the *accept* events in these two rules are not necessarily the same.

Definition 4.12. ***Injective Authentication Contradiction.*** *Given a pair of (not necessarily different) rules* R *and* R' *, it is a contradiction to the injective authentication query* $Q_i = accept \leftarrow[\ B'\ ]\mapsto init, J'$ *denoted as* $Q_i \nvdash \langle R, R' \rangle$ *if and only if (1)* R *and* R' *are obedience rules to* $non\_inj(Q_i)$*; (2) when the corresponding* $init$ *events in* R *and* R' *have the same session id, the* $accept$ *events of* R *and* R' *do not necessarily have the same session id.*

Finally, a rule is a contradiction to the secrecy query when the *leak* event is reachable.

Algorithm 1 Parameter Configuration Computation

Input: $\beta(\mathbb{R}_{init})$ - the rule basis
Input: L_0 - the initial configuration
Input: $\mathbb{Q}_n$ - the non-injective authentication queries
Input: $\mathbb{Q}_i$ - the injective authentication queries
Input: $\mathbb{Q}_i$ - the secrecy queries
Output: $\mathbb{L}$ - a set of parameter configurations
$\mathbb{L} = \{L_0\}$;
for $Q \in \mathbb{Q}_n \cup \mathbb{Q}_s \cup non\_inj(\mathbb{Q}_i), L \in \mathbb{L}, R = [\ G\ ]\ H\ \dashv[\ B\ ]\!\!\rightarrowtail e \in \alpha(\beta(\mathbb{R}_{init}), L)$ **do**
  if $Q \nvdash R$ **then**
    $\mathbb{L} = \mathbb{L} - \{L\}$;
    for $L' : B \cap L' = \emptyset \vee Q \vdash \alpha(R, L')$ **do**
      $\mathbb{L} = \mathbb{L} \cup \{L \cap L'\}$;
    end for
  end if
end for
for $Q \in \mathbb{Q}_i, L \in \mathbb{L}, R = [\ G\ ]\ H\ \dashv[\ B\ ]\!\!\rightarrowtail e\ and\ R' = [\ G'\ ]\ H'\ \dashv[\ B'\ ]\!\!\rightarrowtail e' \in \alpha(\beta(\mathbb{R}_{init}), L)$ **do**
  if $non\_inj(Q) \vdash R \wedge non\_inj(Q) \vdash R' \wedge Q \nvdash \langle R, R' \rangle$ **then**
    $\mathbb{L} = \mathbb{L} - \{L\}$;
    for $L' : B \cap L' = \emptyset \vee B' \cap L' = \emptyset$ **do**
      $\mathbb{L} = \mathbb{L} \cup \{L \cap L'\}$;
    end for
  end if
end for
for $L \in \mathbb{L}, Q \in \mathbb{Q}_n \cup non\_inj(\mathbb{Q}_i)$ **do**
  if $\nexists R \in \alpha(\beta(\mathbb{R}_{init}), L), Q \vdash R$ **then**
    $\mathbb{L} = \mathbb{L} - \{L\}$;
  end if
end for
return $\mathbb{L}$;

Definition 4.13. ***Secrecy Contradiction.*** *A rule $R = [\ G\ ]\ H\ \dashv[\ B\ ]\!\!\rightarrowtail e$ is a contradiction to the secrecy query denoted as $Q_s \nvdash R$ if and only if $G \neq false \wedge B \neq \emptyset$, $e = leak(m)$ and $\forall e' \in H : e' \in \mathbb{V}$.*

During the verification, our goal is to ensure that (1) no contradiction exists for all queries while (2) at least one obedience rule exists for every non-injective authentication query. Hence, given the non-injective authentication queries $\mathbb{Q}_n$, the injective authentication queries $\mathbb{Q}_i$ and the secrecy queries $\mathbb{Q}_s$, our goal is to compute the largest L that

satisfies the following conditions.

$$
\begin{aligned}
(1)\quad & \forall Q \in \mathbb{Q}_n \cup \mathbb{Q}_s \cup non\_inj(\mathbb{Q}_i), \\
& \nexists R \in \alpha(\beta(\mathbb{R}_{init}), L) && : Q \nvdash R \\
(2)\quad & \forall Q \in \mathbb{Q}_i, \nexists R, R' \in \alpha(\beta(\mathbb{R}_{init}), L), \\
& non\_inj(Q) \vdash R, R' && : Q \nvdash \langle R, R' \rangle \\
(3)\quad & \forall Q \in \mathbb{Q}_n \cup non\_inj(\mathbb{Q}_i), \\
& \exists R \in \alpha(\beta(\mathbb{R}_{init}), L) && : Q \vdash R
\end{aligned}
$$

Algorithm 1 illustrates the computing process of the largest L. From line 8 to line 15, we compute the parameter configurations that remove the contradictions for $\mathbb{Q}_n$ and $\mathbb{Q}_s$. From line 16 to line 23, when we find a pair of rules that is a contradiction to an injective authentication query, we remove one of them by updating the global configurations. From line 24 to line 28, we ensure that every non-injective authentication query has at least one obedience rule.

In order to prove the correctness of our algorithm, we need to show that for any configuration L, a contradiction exists in $\alpha(\beta(\mathbb{R}_{init}), L)$ if and only if it exists in $\alpha(\mathbb{R}_{init}, L)$.

Theorem 4.14. ***Partial Correctness.*** *Let $\mathbb{R}_{init}$ be the initial rule set. When Q is a secrecy query or a non-injective authentication query, there exists R derivable from $\alpha(\mathbb{R}_{init}, L)$ such that $Q \nvdash R$ if and only if there exists $R' \in \alpha(\beta(\mathbb{R}_{init}), L)$ such that $Q \nvdash R'$. When Q is an injective authentication query, there exists R_1 and R_2 derivable from $\alpha(\mathbb{R}_{init}, L)$ such that $Q \nvdash \langle R_1, R_2 \rangle$ if and only if there exists $R'_1, R'_2 \in \alpha(\beta(\mathbb{R}_{init}), L)$ such that $Q \nvdash \langle R'_1, R'_2 \rangle$.*

Proof. **Partial Soundness.** Given any rule in $\alpha(\beta(\mathbb{R}_{init}), L)$, according to Theorem 4.10, they are derivable from $\alpha(\mathbb{R}_{init}, L)$. Hence, any contradiction found in $\alpha(\beta(\mathbb{R}_{init}), L)$ is a contradiction derivable from the initial rules $\alpha(\mathbb{R}_{init}, L)$. **Partial Completeness.** (1) When Q is a secrecy query or a non-injective authentication query, suppose we have a rule R derivable from $\alpha(\mathbb{R}_{init}, L)$ such that $Q \nvdash R$. According to Theorem 4.10, R is also derivable from $\alpha(\beta(\mathbb{R}_{init}), L)$. So there exists a derivation tree of R whose nodes

are labeled by rules in $\alpha(\beta(\mathbb{R}_{init}), L)$. We prove that the rule $R_t = [\ G_t\ ]\ H_t\ -[\ B_t\ ]\mapsto e_t$ labeled on the tree's root is also a contradiction as follows. Notice that R is a rule composed by R_t with other rules, so $G_t \neq \mathit{false}$ and $B \neq \emptyset$.

- If Q is a secrecy query, R_t has a leak event as conclusion because $Q \nvdash R$. Additionally, since $R_t \in \alpha(\beta(\mathbb{R}_{init}), L)$, $\forall e'_t \in H_t, e'_t \in \mathbb{V}$. Thus, $Q \nvdash R_t$.

- If $Q = \mathit{accept} \leftarrow[\ B_q\ ]- H_q$ is a non-injective authentication query, e_t should be an accept event. So, R_t should satisfy either $Q \vdash R_t$ or $Q \nvdash R_t$. If $Q \vdash R_t$, as all of the arguments in accept are variables, there exists a substitution σ of e_t and accept satisfying $\sigma\mathit{accept} = e_t$, and $\exists\sigma', (\sigma'\sigma H_q \subseteq \sigma H_t) \wedge (\sigma B_t \subseteq \sigma'\sigma B_q)$ Meanwhile, incoming edges of the tree root cannot be init events and new events, so these events should also persist in R_0. Hence, $Q \vdash R_0$. This violates our precondition that $Q \nvdash R_0$. We then have $Q \nvdash R_t$.

(2) When Q is an injective authentication query, suppose we have a rule pair $\langle R, R'\rangle$ derivable from $\alpha(\mathbb{R}_{init}, L)$ such that $Q \nvdash \langle R, R'\rangle$, in the following we prove that there exists a pair of rules $\langle R_\beta, R'_\beta\rangle$ in $\alpha(\beta(\mathbb{R}_{init}), L)$ such that $Q \nvdash \langle R_\beta, R'_\beta\rangle$. According to Theorem 4.10, R and R' are also derivable from $\alpha(\beta(\mathbb{R}_{init}), L)$. So there exists two derivation trees for R and R' respectively whose nodes are labeled by rules in $\alpha(\beta(\mathbb{R}_{init}), L)$. Suppose the root nodes of these two trees are labeled by R_t and R'_t respectively. We already proved that R_t and R'_t are obedience rules to $\mathit{non\_inj}(Q)$ as above. Given σ is the substitution when the init events are merged in R and R', it should also work when the init events are merged in R_t and R'_t. Because σ cannot merge the accept events in R and R', it cannot merge the accept events R_t and R'_t as well. Hence, we have $Q \nvdash \langle R_t, R'_t\rangle$ □

Checking WMF. After checking the specification of WMF using the above-mentioned algorithm, PTAuth claims an attack. The two key rules in $\beta(\mathbb{R}_{init})$ are shown below. The rule (4.10) represents the execution trace that the server transmits the key once from *Alice* to *Bob*. It is obedient to the query (4.7). However, the rule (4.11) is a contradiction to the query (4.7), because it has a weaker timing range ($t_B \leq t_A + 4 * \S p_a$) than that in the

query ($t_B \leq t_A + 2 * \S p_a$). This rule stands for the execution trace that the adversary sends the message from the server back to server twice and then forwards it to *Alice*. According to the rule (4.5), the timestamp in the message can be updated in this method. Hence, *Bob* would not notice that the message is actually delayed when he receives it. In order to remove the contradiction, we need to configure the parameters as either $\S p_a < \S p_d$ or $\S p_a \leq 0$. However, applying any one of these constraints to the initial configuration $0 < \S p_d$ leads to the removal of the rule (4.10), the only obedience rule in $\alpha(\beta(\mathbb{R}_{init}), L)$. Hence, PTAuth claims that an attack is found, which means that no parameter configuration would make the protocol work.

$$\begin{aligned}
&\mathbf{new}([k], alice_gen[], \langle A[], B[], t_A\rangle), \mathbf{init}([k], \langle A[], B[], [k]\rangle, t_A) \\
&\quad , \mathbf{new}([b], bob_gen[], \langle A[], B[], [k], t_S, t_B\rangle), \mathbf{join}(\langle A[], B[], [k]\rangle, t_S) \\
&\quad \multimap\!\!\!\!\!\!\{\ t \leq t_A, t_B \leq t_S + \S p_a \leq t_A + 2 * \S p_a, t_A + 2 * \S p_d \leq t_S + \S p_d \leq t_B, \}\!\!\!\!\!\!\mapsto \\
&\qquad\qquad \mathbf{accept}([b], \langle A[], B[], [k]\rangle, t_B) \qquad (4.10) \\
&\mathbf{new}([k], alice_gen[], \langle A[], B[], t_A\rangle), \mathbf{init}([k], \langle A[], B[], [k]\rangle, t_A) \\
&\quad , \mathbf{new}([b], bob_gen[], \langle A[], B[], [k], t_{S3}, t_B\rangle), \mathbf{join}(\langle A[], B[], [k]\rangle, t_{S1}) \\
&\quad , \mathbf{join}(\langle B[], A[], [k]\rangle, t_{S2}), \mathbf{join}(\langle A[], B[], [k]\rangle, t_{S3}) \\
&\quad \multimap\!\!\!\!\!\!\{\ t \leq t_A, t_B \leq t_{S3} + \S p_a \leq t_{S2} + 2 * \S p_a \leq t_{S1} + 3 * \S p_a \leq t_A + 4 * \S p_a, \\
&\qquad t_A + 4 * \S p_d \leq t_{S1} + 3 * \S p_d \leq t_{S2} + 2 * \S p_d \leq t_{S3} + \S p_d \leq t_B\ \}\!\!\!\!\!\!\mapsto \\
&\qquad\qquad \mathbf{accept}([b], \langle A[], B[], [k]\rangle, t_B) \qquad (4.11)
\end{aligned}$$

Corrected WMF for Non-injective Timed Agreement. The WMF protocol can be fixed by inserting two different constants m_1 and m_2 into the messages sent to and received from the server respectively, which breaks their symmetric structure. Using this method, the server can distinguish the messages that it sent out previously, and refuse to process them again. Our algorithm proves the non-injective timed agreement of this modified WMF protocol and produces the timing constraints $0 < \S p_d \leq \S p_a$ with the following obedience rule. However, the injective timed agreement of WMF is still unsatisfied.

Corrected WMF for Injective Timed Agreement. In fact, there exist two methods to modify the WMF protocol so that the injective timed agreement can be satisfied.

- In practice, *Bob* can maintain a database which stores the previously used session keys. When a new request is received, *Bob* checks the new session key against the old ones to make sure its uniqueness. Hence, any session key generated by *Alice* can at most correspond to one acceptance by *Bob*. By using the $unique$ event, we can check the uniqueness of values in our calculus.

- According to Lowe's method [82], we can ensure the injective authentication property by adding another round of communications between the protocol initiator and the protocol responder. Before *Bob* engages the $accept$ event in the process P_b, *Bob* can generate a fresh nonce $[n]$ and send it back to *Alice* under the newly agreed encryption key $[k]$. When Alice receives the nonce $[n]$, she send $[n] + 1$ back to *Bob*. Since *Alice* will only reply once, *Bob* then can ensure that his acceptance corresponds to at most one protocol initialization from *Alice*.

Our tool can prove the injective timed agreement property for these two corrected versions of WMF.

4.7 Evaluations

Based on our verification framework, we have implemented a tool named PTAuth. We encode PPL [17] in our tool to analyze the satisfaction of timing constraints. Meanwhile, in order to improve the performance, we implement an on-the-fly verification algorithm that updates the parameter configuration whenever a rule is generated. Hence, the verification process can terminate early if an attack can be found. We use PTAuth to check many security protocols as shown in Table 4.4. All the experiments shown in this section are conducted under Mac OS X 10.10.1 with 2.3 GHz Intel Core i5 and 16G 1333MHz DDR3. In the experiments, we have checked several timed protocols i.e., the WMF protocols [35, 54], the Kerberos protocols [95], the distance bounding protocolse [33, 38, 103] and the CCITT protocols [40, 6, 35]. Additionally, we analyze the untimed protocols like the Needham-Schroeder series [94, 82] and SKEME [72]. As can be seen, most of the

Protocol	Parameterized	Bounded	$\sharp\mathcal{R}$	Result	Time
Wide Mouthed Frog [35]	Yes	No	40	Attack [84]	39ms
Wide Mouthed Frog (c) [54]	Yes	No	35	Secure	13ms
Kerberos V [95]	Yes	No	19370	Attack	23m5s
Kerberos V (c)	Yes	Yes	438664	Secure	2h41m
Auth Range [33, 38]	Yes	No	21	Secure	10ms
Ultrasound Dist Bound [103]	Yes	No	50	Attack [105]	18ms
CCITT X.509 (1) [40]	No	No	45	Attack [6]	14ms
CCITT X.509 (1c) [6]	No	No	62	Secure	37ms
CCITT X.509 (3) [40]	No	No	127	Attack [35]	84ms
CCITT X.509 (3) BAN [35]	No	No	148	Secure	131ms
NS PK [94]	No	No	68	Attack [82]	30ms
NS PK Lowe [82]	No	No	61	Secure	28ms
SKEME [72]	No	No	127	Secure	466ms

Table 4.4: Experiment Results

protocols can be verified or falsified by PTAuth quickly for an unbounded number of protocol sessions. Notice that the secure configuration is given based on the satisfaction of all of the queries, so we do not show the results for different queries separately in the table. The justification for the bounded verification of the corrected version of Kerberos V is presented later in this section. The PTAuth tool and the models shown in this section are available in [2]. Particularly, we have successfully found a new attack in Kerberos V [95] using PTAuth. In the following, we present the detailed findings in Kerberos V. Since Kerberos V is the latest version, we denote it as Kerberos for short unless otherwise indicated.

Kerberos Overview. Kerberos is a widely used security protocol for accessing services. For instance, Microsoft Window uses Kerberos as its default authentication method; many UNIX and UNIX-like operating systems include software for Kerberos authentication. Kerberos has a salient property such that its user can obtain accesses to a network service within a period of time using a single request. In general, this is achieved by granting an access ticket to the user, so that the user can subsequently use this ticket to authenticate himself to the server. Kerberos is complex because multiple ticket operations are supported simultaneously and many fields are optional, which are heavily relying on time. So, configuring Kerberos is hard and error-prone.

Kerberos consists of five types of entities: *User*, *Client*, *Kerberos Authentication Server* (KAS), *Ticket Granting Server* (TGS) and *Application Server* (AP). KAS and TGS together are also known as *Key Distribution Centre* (KDC). Specifically, *Users* usually are humans, and *Clients* represent their identities in the Kerberos network. KAS is the place where a *User* can initiate a logon session to the Kerberos network with a pre-registered *Client*. In return, KAS provides the *User* with (1) a *Ticket Granting Ticket* (TGT) and (2) an encrypted session key as the authorization proof to access TGS. After TGS checks the authorization from KAS, TGS issues two similar credentials (1) a *Service Ticket* (ST) and (2) a new encrypted session key to the *User* as authorization proof to access AP. Then, the *User* can finally use them to retrieve the *Service* from AP. Additionally, both of the TGT and the ST can be postdated, validated and renewed by KDC when these operations are permitted in the Kerberos network.

Specification Highlights. Generally, by following the method described in Section 4.3, the specification for Kerberos itself can be extracted easily. In order to verify Kerberos comprehensively, we model several keys and timestamps (which could be optional) by following precisely its official document RFC 4120 [95].

- The user and the server are allowed to specify sub-session keys in the messages. When a sub-session key is specified, the message receiver must use it to transmit the next message rather than using the default session-key.

- Optional timestamps are allowed in the user requests and the tickets. In the following, fq, tq and rq denote the start-time, the end-time and the maximum renewable end-time requested by the users. Similarly, sp, ep and rp denote the start-time, the end-time and the maximum renewable end-time agreed by the servers. sp, ep and rp are encoded in the tickets, corresponding to fq, tq and rq respectively. An additional timestamp ap is encoded in the ticket to represent the initial authentication time of the ticket. Furthermore, cq represents the current-time when the request is made by the user, and cp stands for the current-time when the ticket is issued by the server. In Kerberos, fq, rq, sp and rp are optional. So the servers need to check their presence and construct replies accordingly.

In this work, two parameters are considered in Kerberos, i.e., the maximum lifetime $\S l$ and the maximum renewable lifetime $\S r$ of the tickets. Based on these parameters, the servers can only issue tickets whose lifetime and renewable lifetime are shorter than $\S l$ and $\S r$ respectively. Furthermore, five operations are modeled for the Kerberos servers as follows. (1) Postdated tickets can be generated for future usage. They are marked as invalid initially and they must be validated later. (2) Postdated tickets must be validated before usage. (3) Renewable tickets can be renewed before they expire. (4) Initial tickets are generated at KAS using user's client. (5) Sub-tickets are generated at TGS using existing tickets. Notice that the end-time ep of the sub-ticket should be no larger than the end-time of the existing ticket. The complete model of Kerberos is available at [2].

Queries. In order to specify the queries, we define three events as follows. Since the injective authentication is not required, we remove the session id encoded in the events for simplicity.

- When an initial ticket is generated at KAS, an $\mathbf{init_{auth}}(\langle k, C, S\rangle, t)$ event is engaged, where k is the fresh session key, C is the client's name, S is the **target** server's name, and t is the beginning of the ticket's lifetime.
- Whenever an new ticket is generated at KAS or TGS, an $\mathbf{init_{gen}}(\langle k, C, S\rangle, t)$ event is engaged. Its arguments have the same meaning as those in $init_{auth}$.
- Whenever an ticket is accepted by the server, an $\mathbf{accept}(\langle k, C, S\rangle, t)$ event is engaged, where k is the agreed session key, C is the client's name, S is the **current** server's name, and t is the acceptance time.

In Kerberos, we need to ensure the correctness of two timed authentications. First, whenever a server accepts a ticket, the ticket should be indeed generated within $\S l$ time units using the same session key. Second, whenever a server accepts a ticket, the initial ticket should be indeed generated within $\S r$ time units.

$$accept(\langle k, C, S\rangle, t) \leftarrow\!\!\lfloor\ t - t' \leq \S l\ \rfloor\!\!\vdash\ init_{gen}(\langle k, C, S\rangle, t') \tag{4.12}$$

$$accept(\langle k, C, S\rangle, t) \leftarrow\!\!\lfloor\ t - t' \leq \S r\ \rfloor\!\!\vdash\ init_{auth}(\langle k', C, S'\rangle, t') \tag{4.13}$$

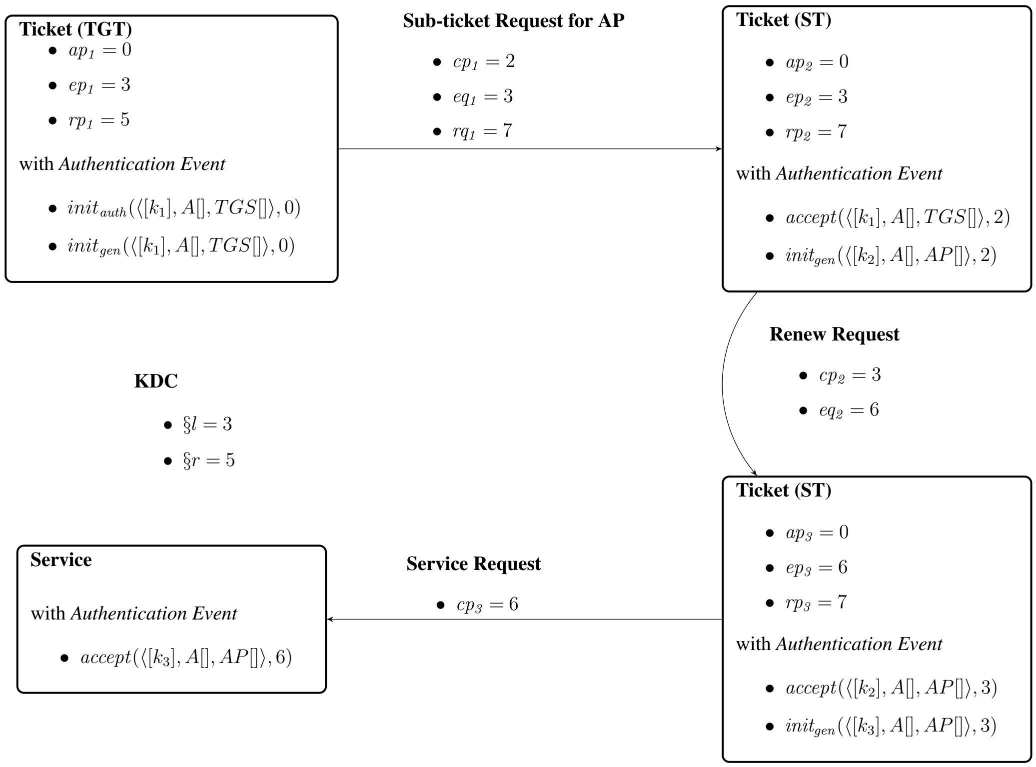


FIGURE 4.1: Attack Found in Kerberos V

Verification Results. For the termination of the verification, we need to initially configure the parameters as $\S r < n * \S l$, where n can be any integer larger than 1. The requirement for this constraint is justified as follows. Algorithm 1 updates parameter configuration at line 15 to eliminate the contradiction rules. Suppose we have a rule $init_{auth}(\langle k, C, S \rangle, t') \dashv t - t' \leq c * \S l \mapsto accept(\langle k, C, S \rangle, t)$ in the rule basis, where $c > 1$. This rule is a contradiction to the query (4.13) because $\S r$ is not necessarily larger than $c * \S l$. However, Algorithm 1 can add a new constraint $c * \S l \leq \S r$ to the existing configuration and then continue searching. Since we have infinitely many such rules in $\beta(\mathbb{R}_{init})$ with different values of c, the verification cannot terminate. Hence, in this work, we set the initial configuration as $\S r < 2 * \S l$ to avoid the non-termination. Notice that this initial configuration does not prevent us from finding attacks because it does not limit the number of sequential operations allowed in the Kerberos protocol.

After analyzing Kerberos using PTAuth, we have successfully found a security flaw in its specification document RFC 4120 [95]. (When the network latency is not considered,

PTAuth directly reports the attack; when the network latency is considered, PTAuth claims that the protocol is correct only if $l \leq 2 * \S p_d$, which is clearly undesired.) The attack trace is depicted in Figure 4.1. Suppose the Kerberos is configured with $\S l = 3$ and $\S r = 5$[2], and a user Alice has already obtained a renewable ticket at time 0. Then, she can request for a sub-ticket of AP at time 2 that is renewable until time 7, satisfying $rq_1 - cp_1 \leq \S r$. Notice the new sub-ticket's end-time ep_2 cannot be larger than the end-time ep_1 of the existing ticket. Later, she renews the new sub-ticket before it expires and gets a ticket valid until time 6. Finally, she requests the service at time 6 and engages an event $accept(\langle [k_3], A[], AP[] \rangle, 6)$. However, this accept event does not correspond to any $init_{auth}$ event satisfying Query (4.13), which leads to an attack. In fact, Alice can use this method to request sub-ticket for AP repeatedly so that she can have access to the service forever. Obviously, the server who made the authentication initially does not intend to do so. Fortunately, after checking the source code of Kerberos, we find that this flaw is prevented in its implementations [90, 74]. An additional checking condition[3] has been inserted to regulate that the renewable lifetime in the sub-ticket should be smaller than the renewable lifetime in the existing ticket. We later confirmed with Kerberos team that this is an error in its specification document, which could have led to a security issue but has not done so in its current implementation.

Corrected Version. After adding the timing constraints on renewable lifetime between the base-ticket and the sub-ticket, the verification cannot terminate. This is caused by an infinite dependency trace formed by tickets, as we do not limit its length. Hence, we bound the number of tickets that can be generated during the verification, which in turn bounds the number of $init_{gen}$ events in the rule. In this work, we bound the ticket number to five. This is justified as we have five different methods to generate tickets in Kerberos: the servers can postdate, validate, renew tickets, generate initial tickets and issue sub-tickets. After bounding the ticket number that can be generated, our tool proves the correctness of Kerberos and produces the configuration $0 \leq \S l \leq \S r < 2 * \S l$.

[2] $\S l$ and $\S r$ are represented by symbols during the verification.

[3] For krb5-1.13 from MIT, the checking is located in the file src/kdc/kdc_util.c at line 1740 - 1741. We also checked other implementations, like heimdal-1.5.2.

4.8 Related Works

As mentioned, this work is related to the work [77] shown in Chapter 3. In this work, we additionally introduce timing parameters, secrecy and injective authentication properties, and enhance the computation capability of the timing constraint with PPL. Furthermore, we provide the algorithm to compute the least constrained secure configuration of parameters in this work. We successfully analyze several protocols including Kerberos V and find an attack in the Kerberos V specification [95] that is unreported before. The analyzing framework closest to ours was proposed by Delzanno and Ganty [54] which applies $MSR(\mathcal{L})$ to specify unbounded crypto protocols by combining first order multiset rewriting rules and linear constraints. According to [54], the protocol specification is modified by explicitly encoding an additional timestamp, representing the initialization time, into some messages. Thus the attack can be found by comparing the original timestamps with the new one in the messages. However, it is unclear how to verify timed protocol in general using their approach. On the other hand, our approach can be applied to protocols without any protocol modification. Many tools for verifying protocols [29, 48, 89] are related. However, they are not designed for timed protocols.

Kerberos has been scrutinized over years using formal methods. In [27], Bella et al. analyzed Kerberos IV using the Isabelle theorem prover. They checked various secrecy and authentication properties and took time into consideration. However, Kerberos is largely simplified in their analysis and the specification method in their work is not as intuitive as ours. Later, Kerberos V has been analyzed by Mitchell et al. [91] using state exploration tool Murφ. They claimed that an attack is found in [70] when two servers exists. However, this attack is actually prevented in Kerberos's official specification document RFC 1510 [69], which is later superseded by RFC 4120 [95] analyzed in this work. The biggest advantages of our method is that the verification is given for an unbounded number of sessions, which is not achievable previously with the state exploration approach. For the above literatures, they did not consider alternative options supported in the protocol that may accidentally introduce attacks as we do in this work. Similar to our work, Kerberos V has been analyzed in a theorem proving context by Butler et al. [36]. They took many fea-

tures into consideration, i.e., the error messages, the encryption types and the cross-realm support. These features are not cover in this work since we focus on the timestamps and timing constraint checking. Meanwhile, our framework can provide intuitive modeling and automatic verifying, while Kerberos V is analyzed manually in [36].

4.9 Discussions

In this work, we developed an automatic verification framework for timed parameterized security protocols. It can verify authentication properties as well as secrecy properties for an unbounded number of protocol sessions. We have implemented our approach into a tool named PTAuth and used it to analyze a wide range of protocols shown in Section 4.7. In the experiments, we have found a timed attack in Kerberos V document that has never been reported before.

Since the problem of verifying security protocols is undecidable in general, we cannot guarantee the termination of our verification algorithm. When we use PTAuth to analyze the corrected version of Kerberos, PTAuth cannot terminate because of the infinite dependency chain of tickets. Hence, we have to bound the number of tickets generated in the protocol. However, in Kerberos, generating more tickets may not be helpful to break its security. Based on this observation, we want to detect and prune the non-terminable verification branches heuristically without affecting the final results in our future work. This could help us to verify large-sized and complex protocols that we cannot verify currently, as our verification algorithm only considers the general approach at present.

Chapter 5

Analyzing Software-based Attestation in Practice

An increasing number of "smart" embedded devices are employed in our living environment. Unlike traditional computer systems, these devices are often physically accessible to the attackers. It is therefore almost impossible to guarantee that they are uncompromised, i.e., that indeed the devices are executing the intended software. In such a context, software-based attestation [108, 110, 106] is deemed as a promising solution to validate their software integrity. It guarantees that the softwares running on the embedded devices are un-compromised without any hardware support. However, designing software-based attestation protocols has been shown as error-prone [111, 39]. In this chapter, we develop a framework to design and analyze the software-based attestation protocols. We first propose a generic attestation scheme that captures most existing software-based attestation protocols. After formalizing the security criteria for the generic scheme, we apply our analysis framework to several well-known software-based attestation protocols and report various potential vulnerabilities. To the best of our knowledge, this is the first practical analysis framework for software-based attestation protocols.

5.1 Introduction

"Smart" sensory embedded devices are getting more and more popular. They are frequently used for temperature measurement, fire detection, water saving, etc. In the near future, they are expected to be ubiquitous. However, their wide adoption poses threats to our safety and privacy as well. Unlike traditional computer systems, these devices are often physically accessible to the attackers and it is almost impossible to guarantee that they are un-compromised, i.e., that indeed the devices are executing the intended software. Effective techniques for verifying and validating the embedded devices against malicious adversary becomes increasingly important and urgent. Traditional hardware-based attestation [14, 58, 102, 67] is cost-ineffective in such a context. Thus, software-based attestation [108, 110, 106], which aims to function without any dedicated security hardware, is deemed as a promising solution for verifying the integrity of these massive, inexpensive, and resource constrained devices.

Software-based attestation is based on the challenge-response paradigm between the trusted verifier and the potentially compromised prover (the embedded device). It typically works as follows. The verifier first sends a random challenge to the prover and asks the prover to generate a checksum for its memory state based on the challenge. Since the prover's computing and memory resources are designed to be fully utilized in the attestation, if the memory is tampered by the adversary, the prover needs to take extra time to compute the correct checksum. We further assume that the verifier knows the expected memory state of the prover. He thus can compute the same checksum and compare it with the one received from the prover. By exploiting the fact that the prover is resource constrained, software-based attestation ensures that the prover can return the correct response in time only if it is genuine. On the other hand, whenever the prover fails to reply in time or returns an incorrect checksum, it is highly likely compromised.

The software-based attestation protocol design is challenging and error-prone [111,

39]. Hence, in this work, we propose an analysis framework for software-based attestation that can be easily adopted in practice. First, our framework provides a parameterized generic software-based attestation scheme that captures most existing software-based attestation protocols. The adversary modeled in this work can not only compromise the prover before the attestation, but also communicate with the compromised prover during the attestation. We then formalize the security criteria for the generic scheme based on the knowledge of network latency (which is important as timing is essential here) and adversary model. Since the real software-based attestation protocols are instances of the generic scheme, these criteria thus naturally should be hold in the real protocols as well. Hence, we apply our analysis framework to three well-known software-based attestation schemes, i.e., SWATT [108], SCUBA [106] and VIPER [79], and find four potential vulnerabilities that have not been reported before. As far as we know, this is the first framework that can give practical analysis to real software-based attestation protocols.

5.2 Generic Specification for Software-based Attestation

We start with defining a generic software-based attestation scheme which captures most existing software-based attestation protocols. The idea is that analysis results based on the generic schema can be extended to concrete protocols readily as we show in later sections. The generic software-based attestation scheme involves three parties, i.e., the trusted verifier $\mathcal{V}$, the prover (the embedded device) $\mathcal{P}$ and the adversary $\mathcal{A}$. We denote the genuine prover and the compromised prover as $\mathcal{P}_g$ and $\mathcal{P}_c$ respectively. In this section, we first present the system model, including the system architecture, the security property and the threat model. Then we propose a generic software-based attestation scheme between the trusted verifier $\mathcal{V}$ and the genuine prover $\mathcal{P}_g$ based on our system.

5.2.1 System Overview

Software-based attestation is proposed to verify the resource constrained embedded devices without using any security hardware (e.g., TPMs [3]). Before presenting the details of the generic attestation scheme, we first describe the system model employed in this work. The attestation procedure is conducted between a trusted verifier $\mathcal{V}$ and a prover $\mathcal{P}$ over the network. We explicitly consider the network round-trip time (RTT).

The architecture of the verifier $\mathcal{V}$ and the prover $\mathcal{P}$ considered in this work are depicted as follows. $\mathcal{P}$ consists of a computing processor, several registers and a memory M. The data memory M_d and the program memory M_p are two different memory space that should be attested in M. Specifically, M_d stores the runtime data (e.g., stack information, data collected from the environment) that are unpredictable to $\mathcal{V}$, hence its content cannot be attested directly in the attestation procedure. M_p stores the program code which is known to $\mathcal{V}$. The attestation routine $verif$ on the prover side is pre-installed in M_p before the attestation starts. In general, the size of M_d could be 0 when the attestation for the data memory is not required. Notice that some memory can be excluded from the attestation in some specific attestation protocols [107, 106, 79], and thus $M_d + M_p$ may not equal to M. Meanwhile, $\mathcal{V}$ is a powerful base station who can simulate the execution of $\mathcal{P}$. When $\mathcal{V}$ has the image of both M_d and M_p in $\mathcal{P}$, $\mathcal{V}$ can compute the memory checksum based on the image.

During the attestation, $\mathcal{P}$'s data memory M_d will be first overwritten into a state that is known to $\mathcal{V}$. The attestation then aims at verifying whether $\mathcal{P}$ has a genuine state for both M_d and M_p as $\mathcal{V}$ expected. Let $State(\mathcal{P})$ be the memory state of $M_d + M_p$ in the prover $\mathcal{P}$. When $State(\mathcal{P})$ is known to $\mathcal{V}$, the attestation can be modeled by a game between the verifier $\mathcal{V}$ and the prover $\mathcal{P}$. In the game, $\mathcal{V}$ first sends a random challenge to $\mathcal{P}$, and then $\mathcal{P}$ picks a checksum reply based on the challenge. The prover $\mathcal{P}$ wins if the used time is less than some threshold and the checksum is correct, other-

wise $\mathcal{P}$ loses the game. We denote the percentage of differences between two memory states S and S' as $\lambda(S, S')$ and the winning probability of $\mathcal{P}$ as $\mathbb{P}_w(\mathcal{L}, \mathcal{P})$, where $\mathcal{L}$ denotes the system and its configurations. We define an attestation protocol as *correct* if $\mathbb{P}_w(\mathcal{L}, \mathcal{P}_g) = 1$, which means that the genuine prover $\mathcal{P}_g$ can always win. On the other hand, when μ is the least memory proportion that should be modified in the compromised prover $\mathcal{P}_c$ to perform a meaningful attack, we define an attestation protocol as $\langle\varepsilon, \mu\rangle$-*secure* if $\forall \mathcal{P}_c, \lambda(State(\mathcal{P}_c), State(\mathcal{P}_g)) \geq \mu > 0 \Rightarrow \mathbb{P}_w(\mathcal{L}, \mathcal{P}_c) \leq \varepsilon$, which means that any prover who needs to overwrite at least μ percentage of the attested memory has the winning probability of no more than ε. In the attestation, the adversary wins if and only if he can keep the malicious code in the attested memory after the attestation. However, software-based attestation does not guarantee that the device is unmodified before the attestation.

The adversary $\mathcal{A}$'s capability is specified with two phases. Before the attestation begins, $\mathcal{A}$ can use unlimited resources to reprogram the memory in $\mathcal{P}_c$. However, $\mathcal{A}$ cannot change the physical hardware and the network infrastructure, so $\mathcal{P}_c$'s memory storage, computing power and network latency are fixed. Once the attestation starts, $\mathcal{A}$ cannot modify $\mathcal{P}_c$'s memory content anymore. Nevertheless, $\mathcal{A}$ can communicate with $\mathcal{P}_c$ over the network and compute with unlimited resources.

Notations. The notations used in this chapter are listed as follows. We write X, Y, Z to denote sets and x, y, z to denote elements in the sets. $f(x : X, y : Y) \to z : Z$ represents a function f that maps the tuple of two elements x, y to the element z. Let n be a natural number. X^n stands for the concatenation of n elements in X. $X \times Y$ is the Cartesian product of X and Y. Let $\mathbb{D}$ be a probabilistic distribution over set X. $x \leftarrow\!\!\dashv \mathbb{D} \vdash X$ means assigning an element of X to x according to $\mathbb{D}$. $[n \ldots m]$ represents the integers from n to m. $[n, m]$ stands for the real numbers from n to m. $max_{x,y}\{f(x, y)\}$ stands for the maximum value of $f(x, y)$ for any x and y. $Pr[x]$ denotes the probability of x.

Checksum Computation $comp(S_a, g_0, r_0)$
S_a is the memory state of $\mathcal{P}$ under attestation.
g_0 is the address generator seed.
r_0 is the checksum response seed.

for i **in** $[1 \dots n]$ **do**
  $g_i = Gen(g_{i-1})$;
  $a_i = Addr(g_i)$;
  $c_i = Read(S_a, a_i)$;
  $r_i = Chk(r_{i-1}, c_i)$;
end
return r_n;

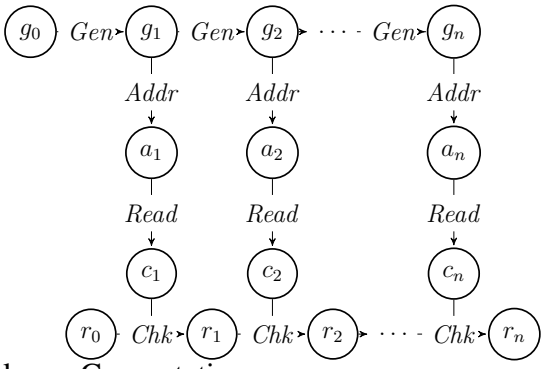


FIGURE 5.1: Checksum Computation

5.2.2 Generic Attestation Scheme

In this section, we propose a generic specification for software-based attestation scheme that captures most existing software-based attestation protocols. The specification is described in two parts. First, given a memory state $S_a = State(\mathcal{P})$ of both M_d and M_p, we introduce the *checksum computation routine* that compute the memory checksum as shown in Figure 5.1. Then, we illustrate the *generic software-based attestation scheme* which first securely erases the data memory M_d and then attests the whole memory $M_d + M_p$ with the *checksum computation routine*.

The checksum computation routine $comp(S_a, g_0, r_0)$ aims at computing the unforgeable checksum for memory state S_a based on the initial address generator g_0 and initial memory checksum r_0. It iteratively computes the address generator g_i, the memory address a_i, the memory content c_i and the checksum response r_i for $i \in [1 \dots n]$ as shown in Figure 5.1. The four functions used in the generic scheme are illustrated as follows. In the following chapter, l_g, l_a, l_c and l_r represent lengths of g_i, a_i, c_i and r_i respectively.

- $Gen(g_{i-1} : \{0,1\}^{l_g}) \rightarrow g_i : \{0,1\}^{l_g}$ computes the generator g_i of the memory addresses in a random manner incrementally.
- $Addr(g_i : \{0,1\}^{l_g}) \rightarrow a_i : \{0,1\}^{l_a}$ converts the random generator g_i to the memory address a_i.

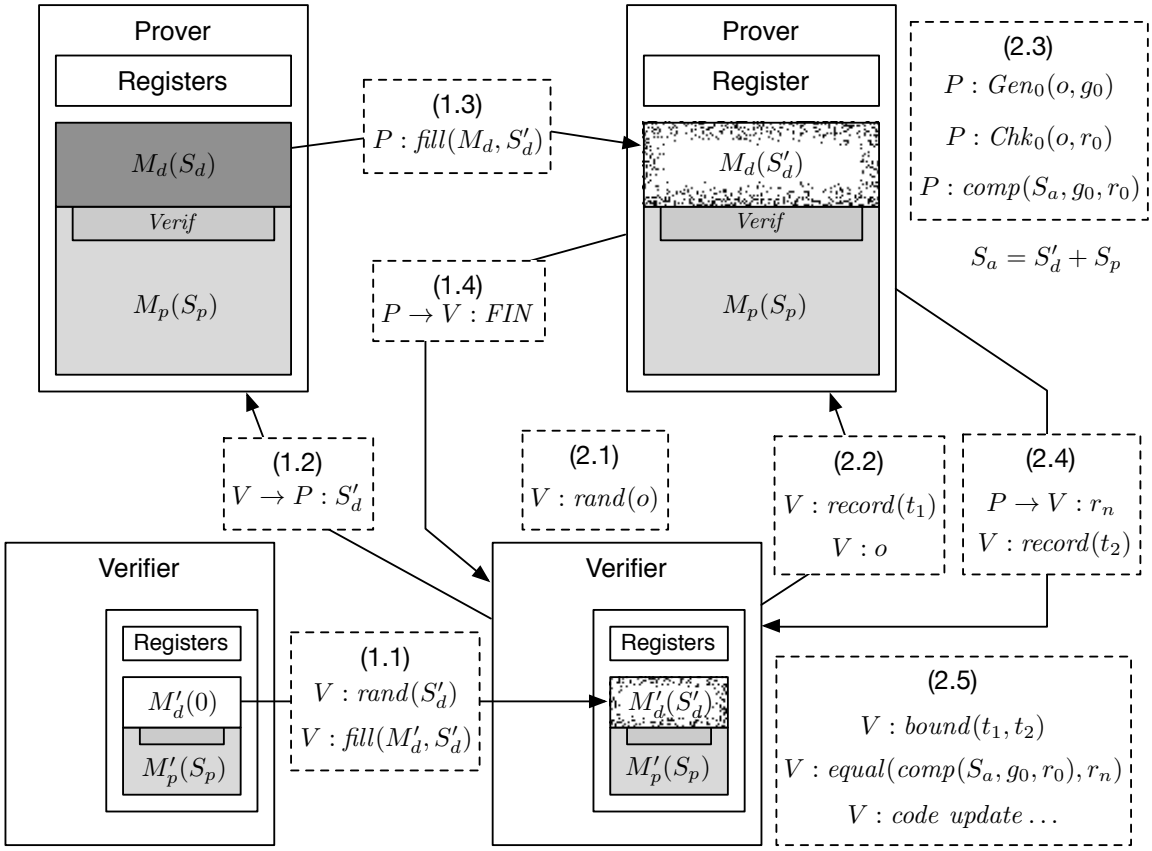


FIGURE 5.2: Generic Software-based Attestation Scheme

- $Read(S_a : \{0,1\}^{l_a} \times \{0,1\}^{l_c}, a_i : \{0,1\}^{l_a}) \rightarrow c_i : \{0,1\}^{l_c}$ reads the memory content c_i located at the address a_i in S_a.

- $Chk(r_{i-1} : \{0,1\}^{l_r}, c_i : \{0,1\}^{l_c}) \rightarrow r_i\{0,1\}^{l_r}$ updates the last checksum response r_{i-1} with the memory content c_i to the new checksum r_i.

The generic software-based attestation scheme is shown in Figure 5.2. The functions used in the figure are illustrated as follows. $rand(x)$ generates a random bit-string and stores it into x. $fill(M, S)$ fills the memory M with state S. $Gen_0(o, g_0)$ and $Chk_0(o, r_0)$ derive the initial values for the generator and the checksum from the challenge o and store them into g_0 and r_0 respectively. $comp(S_a, g_0, r_0)$ illustrated previously computes the checksum for memory state S_a with the generator seed g_0 and the response seed r_0.

$record(t)$ records the current time into t. $bound(t_1, t_2)$ checks whether $t_2 - t_1$ is smaller than a time bound. $equal(x, y)$ checks if x and y are equivalent. $I : op$ means that I conducts the operation op. $I_1 \rightarrow I_2 : m$ means that I_1 sends the message m to I_2. The generic software-based attestation scheme proposed in this work is divided into two phases as shown in Figure 5.2.

Phase 1. Secure Erasure overwrites the data memory M_d with random noise. Initially, $\mathcal{P}$'s data memory image M'_d in $\mathcal{V}$ are filled with 0, while M_d in $\mathcal{P}$ has the memory state S_d consisting of information generated at runtime. At the end of this phase, $\mathcal{P}$ and $\mathcal{P}$'s image in $\mathcal{V}$ have the same memory state S'_d filled with random noise.

1. When $\mathcal{V}$ wants to start the attestation, it first overwrites $\mathcal{P}$'s data memory image M'_d in $\mathcal{V}$ to a random state S'_d, which is generated by the $rand(S'_d)$ function.
2. $\mathcal{V}$ sends S'_d to $\mathcal{P}$ and asks $\mathcal{P}$ to overwrite its M_d with S'_d.
3. $\mathcal{P}$ accepts $\mathcal{V}$'s requests and updates his M_d with S'_d. In fact, the last step (1.2) and this step (1.3) can be streamlined. Whenever $\mathcal{P}$ receives a value from $\mathcal{V}$, he writes it into the corresponding data memory location.
4. When M_d is filled with S'_d, $\mathcal{P}$ sends a FIN signal to start the second phase.

Phase 2. Checksum Computation aims at attesting both M_d and M_p in $\mathcal{P}$ and discovering memory modification with overwhelming probability. When the first phase is finished, $\mathcal{V}$ can run the second phase for multiple times consecutively. Upon the beginning of the second phase, $\mathcal{V}$ knows the memory state $S_a = State(\mathcal{P})$.

1. $\mathcal{V}$ first picks a random challenge o.
2. $\mathcal{V}$ sends o to $\mathcal{P}$ and asks $\mathcal{P}$ to compute the checksum for his memory state $S_a = S_p + S'_d$. $\mathcal{V}$ also records the time t_1 when the request is sent.

3. After $\mathcal{P}$ derives the initial address generator g_0 and the initial checksum response r_0 from the challenge o, he computes the checksum over the memory state S_a with $comp(S_a, g_0, r_0)$ illustrated in Figure 5.1.

4. As soon as the checksum computation routine is finished, $\mathcal{P}$ sends the checksum r_n back to $\mathcal{V}$. $\mathcal{V}$ again records the time t_2 when r_n is received.

5. Once $\mathcal{V}$ receives r_n from $\mathcal{P}$, he checks two conditions: (1) whether the checksum is received within the timing threshold $\{bound(t_1, t_2) = true\}$ and (2) whether the checksum is correct $\{equal(comp(S_a, g_0, r_0), r_n) = true\}$. If both of the conditions are satisfied, $\mathcal{P}$ is trusted as genuine and $\mathcal{V}$ will update $\mathcal{P}$'s unattested memory. Otherwise, $\mathcal{P}$ is deemed as compromised.

Adversary Model. The attacker has full control over the memory of the device. However, the attacker cannot modify the hardware of the device and increase the computation power of the device. For instance, the attacker cannot increase the size of the memory with new memory cards; the attacker cannot increase the clock speed of the processor with the BIOS settings.

Assumptions. In order to guarantee the correctness of the protocol, we make the following assumptions. First, $\mathcal{P}$ either has the attestation procedure $verif$ pre-deployed in its program memory M_p or can download it into a pre-allocated memory space in M_p at runtime before the attestation starts. Second, $\mathcal{V}$ knows the exact memory image of M_p in $\mathcal{P}$. M_d and M_p share the same address space. Third, the attestation procedure $verif$ implemented in $\mathcal{P}$ is optimal in terms of execution speed. Fourth, S'_d and o are unpredictable to the prover. Fifth, the cryptographic primitives used in the attestation procedure are perfect. This assumption does not reduce the security offered by our framework to the real applications. We can update the attestation procedure with the state-of-the-art cryptographic implementations that are unbreakable at the moment. For instance, when a hash function is needed in the attestation, we use *SHA-2* or *SHA-3* that are safe for the time

being. Sixth, the adversary cannot personate the prover and communicate with the verifier directly, which means that the verifier is connected to the prover via a controllable channel during the attestation, e.g., a *bus* used in [79]. When the adversary can personate the prover, the software-based attestation protocol is trivially broken because the adversary can answer the challenge for the prover.

5.3 Security Criteria Formalization

In this section, we introduce several attack scenarios. Based on the attacks, we formalize the security criteria for the generic attestation scheme. When the compromised prover $\mathcal{P}_c$ computes the checksum by itself, we need to discuss two cases: (1) the checksum is computed with the checksum computation routine at runtime, or (2) the checksum is pre-computed. In the first case, when the memory and the registers are fully utilized as shown in Section 5.3.1, we measure the winning probability of $\mathcal{P}_c$ who trades computation power for memory space (*memory recovering attack*) in Section 5.3.2. In the second case, we discuss the scenario where $\mathcal{P}_c$ stores the pre-computed challenge-response pairs in the its memory (*challenge buffering attack*) in Section 5.3.3. On the other hand, when $\mathcal{P}_c$ does not compute the checksum by itself, it can ask $\mathcal{A}$ to compute the checksum (*proxy attack*) as introduced in Section 5.3.4. When the memory and the registers are fully attested, since the above three attack methods are orthogonal, the winning probability of the compromised prover $\mathbb{P}_w(\mathcal{L}, \mathcal{P}_c)$ then can be calculated by the most effective attack among them. Some used notations are summarized in Table 5.1.

5.3.1 Full Utilization of Memory and Registers

In the checksum computation routine, the memory are accessed in a random manner which is unpredictable for the prover before the attestation. Whenever the attested memory is tampered, the malicious prover thus need to take extra time to recover the original memory. In order to prevent the malicious prover from cheating, every memory address should

Name	Explanation	Size
$M_d(S_d)$	Data memory M_d filled with memory image state S_d	m_d unit
$M_p(S_p)$	Program memory M_p filled with memory image state S_p	m_p unit
$M(S)$	Overall memory M filled with memory image state S	m unit
o	The challenge sent from $\mathcal{V}$ to $\mathcal{P}$	l_o bit
g_i	Address generators for $i \in [0 \dots n]$	l_g bit
a_i	Memory addresses for $i \in [0 \dots n]$	l_a bit
c_i	Memory contents for $i \in [0 \dots n]$	l_c bit
r_i	Checksum responses for $i \in [0 \dots n]$	l_r bit
$T_{\mathcal{V}}^{min}, T_{\mathcal{V}}^{max}$	Network RTT between $\mathcal{V}$ and $\mathcal{P}_g$ varies from d_g^{min} to d_g^{max}	-
$T_{\mathcal{A}}^{min}, T_{\mathcal{A}}^{max}$	Network RTT between $\mathcal{A}$ and $\mathcal{P}_c$ varies from d_c^{min} to d_c^{max}	-
$d_{Gen}, d_{Addr}, d_{Read}, d_{Chk}$	Computation time for Gen, $Addr$, $Read$ and Chk resp.	-
d_g	The time needed by $\mathcal{P}_g$ to compute the memory checksum	-
d_{th}	The timing threshold on the verifier side	-
n	The number of iterations in a single checksum computation	-
k	The number of consecutive checksum computation (Phase 2)	-
u	The number of registers used to store the checksum	-

Table 5.1: Notation Summary

be accessible in the checksum computation. Additionally, the registers should be fully occupied as well. In this section, we formalize several design principles to ensure fully utilization of the memory and registers in the checksum computation routine.

Choosing Random Function. During the checksum computation, Gen is a random function from l_g bits to l_g bits, and $Addr$ converts the l_g bit generators to the l_a bit addresses. Thus, we can take the concatenation of Gen and $Addr$ as a random function from l_g bits to l_a bits. Since all possible addresses should be accessible when the generators are traversed, proper configuration of the random function in the attestation scheme becomes non-trivial. We discuss two kinds of randomization functions in this work, i.e., the *hash oracle* and the *encryption oracle*.

The *hash oracle* receives a bit-string as input and returns a corresponding random bit-string as output. Since every hash output is computed independently, according to the *coupon collector's problem*, the expected number of independent runs to cover all possible output values grows as $\Theta(t \cdot log(t))$ where t is the number of possible output values. In other words, if the addresses (a_i) and the generators (g_i) have the same

length, it is very likely that some memory addresses are uncovered. For instance, when the hash function *SHA-2* is used and both of the generator and the memory address have the same length of 32bit, only 64% of the addresses can be covered on average when the generators are traversed in our experiments. By enumerating all possible generators in the preparation phase, the adversary may find sufficient *uncovered* addresses and use them to store the malicious code. As a consequence, when hash oracle is used in the attestation protocols, the number of generators should be much larger than the number of addresses. By applying the tail estimate to the *coupon collector's problem*, we can calculate the probability lower-bound of covering all addresses under attestation as $1-(m_d+m_p)^{1-2^{lg}/((m_d+m_p)\cdot log(m_d+m_p))}$.

On the other hand, the *encryption oracle* can be used to generate random numbers as well by revealing the encryption key to the public. Since the encryption oracle is bijective, all of the memory addresses should be covered in the generator traversal when the generator length is not less than the address length. As a result, the *encryption oracle* becomes very suitable for the random number generation in software-based attestation. Two heavily used implementations of the encryption oracle in the software-based attestation protocols are the stream cipher RC4 and the T-function [68]. RC4 is chosen as the PRNG in SWATT [108] because of its extreme efficiency and compact implementation in the embedded devices. Meanwhile, T-function can produce a single cycle, which ensures the traversal of generators. Thus, it is employed in ICE scheme proposed in ICUBA [106]. A widely used T-function is $x \leftarrow x + (x^2 \vee 5)$ where $\vee$ is the bitwise *or* operator.

Full Address Coverage at Runtime. Even though the addresses can be fully covered in the generator traversal, the actual address coverage is also related to the number of addresses generated at the runtime, which is decided by the number n in the checksum computation routine (Figure 5.2) and the repeat time k of the consecutive checksum computation (Phase 2). According to the *coupon collector's problem*, in order to fully traverse

the whole memory space in the attestation procedure, the minimal number of memory access $n \cdot k$ should satisfy

$$Pr[n \cdot k > c \cdot (m_d + m_p) \cdot log(m_d + m_p)] \leq (m_d + m_p)^{1-c}. \tag{5.1}$$

Full Register Occupation. According to several existing works [108, 106, 79], the registers in $\mathcal{P}$ are frequently used to store the checksum results. During every iteration in the checksum computation, one of them gets updated to a new value. When any register is unused in the attestation, the malicious prover can exploit it to conduct attacks. Thus, all the registers should be occupied. Moreover, the registers should be chosen in a random order so the malicious prover cannot predict which one is used next. Let the total number of registers used for storing the checksum be u. According to the *coupon collector's problem*, the probability of covering all registers in the checksum computation is lower-bounded by $1 - u^{1-n/(u \cdot log(u))}$.

5.3.2 $\mathcal{P}_c$ Compute Checksum at Runtime: Memory Recovering Attack

Given a genuine prover $\mathcal{P}_g$ with the memory state S_g and a compromised prover $\mathcal{P}_c$ with the memory state S_c, the probability of distinguishing their states with a single memory access depends on two factors. The first factor is the percentage of the differences between S_g and S_c, which could be defined as $\lambda(S_g, S_c) = Pr[Read(S_g, a) \neq Read(S_c, a) | a \in \{0,1\}^{l_a}]$. When $\lambda(S_g, S_c)$ is sufficiently large, we can easily detect the modifications in the memory. The second factor is related to the memory content bias in $\mathcal{P}_g$. For instance, the program in $\mathcal{P}_g$ usually contains a large amount of duplicated assembly code such as *mov*, *jmp*, *call*, *cmp*, *nop*, etc. These assembly code can be approximated with high probability. As a consequence, the compromised prover can overwrite the biased memory content into malicious code and recover the original content using a recovering algorithm $\mathcal{C}$ with high probability. Assume the overwriting algorithm is $\mathcal{W}$, the minimal overwriting potion is μ, and memory recovering time $d_{\mathcal{C}}$ is no more than $\delta \cdot d_{Read}$ as required, we could calculate

the optimal success probability of the memory recovery as

$$\mathbb{P}_m(S,\mu,\delta) = max_{\mathcal{C},\mathcal{W}}\{Pr[Read(S,a) = \mathcal{C}(\mathcal{W}(S),a) \\ \mid a \in \{0,1\}^{l_a}] \mid \delta \cdot d_{Read} \geq d_{\mathcal{C}} \wedge \lambda(S,\mathcal{W}(S)) \geq \mu\}$$

for any recovering algorithm $\mathcal{C}$ and overwriting algorithm $\mathcal{W}$. δ is the allowed timing overhead for the recovering algorithm comparing with the $Read$ operation. We will discuss more about δ in Section 5.3.4. When $\delta \geq 1$, we can always implement the recovering algorithm $\mathcal{C}$ for any S as $\mathcal{C}(S,a) = Read(S,a)$, so $\mathbb{P}_m(S,\mu,\delta) \geq 1-\mu$.

Since $\mathcal{P}_c$ needs to recover the memory content for n times in the checksum computation routine, he can compute the correct checksum if either the memory is recovered successfully for every iteration or the computed checksum collides with the correct one. So overall success probability for $\mathcal{P}_c$ is $\mathbb{P}^n_m(S,\mu,\delta) + (1-\mathbb{P}^n_m(S,\mu,\delta)) \cdot 2^{-l_r}$. As can be seen from the formula, the success probability is lower-bounded by 2^{-l_r}. So increasing n becomes less significant when n becomes larger. As a consequence, we can define a threshold η for the potential probability increase and then give a lower-bound to the n used in the checksum computation.

$$\mathbb{P}^n_m(S,\mu,\delta) \cdot (1-2^{-l_r}) \leq \eta \quad \Longrightarrow \quad n \geq \frac{log(\eta) - log(1-2^{-l_r})}{log(\mathbb{P}_m(S,\mu,\delta))} \tag{5.2}$$

In this work, we suggest to set $\eta = 2^{-l_r}$ which is the success probability's lower-bound. Additionally, we recommend the attestation protocols to set n as the lower-bound given in formula (5.2) for efficiency and conduct the checksum computation phase (Phase 2) for multiple times to give better security guarantee.

Full Randomization of Data Memory. In the first phase of the generic attestation scheme, $\mathcal{V}$ asks $\mathcal{P}$ to overwrite its data memory with S'_d filled with noise. The unpredictability of S'_d enforces $\mathcal{P}$ to erase its data memory completely. A similar design is taken in [44], but its S'_d is generated by $\mathcal{P}$ using a PRNG seeded by a challenge sent from $\mathcal{V}$. As we discussed above, the recovering algorithm can use the PRNG to generate the memory state with the

received challenge at runtime, so $\mathcal{P}_c$ can trade the computation time for memory space. As a result, $\mathcal{P}_c$ can keep the malicious code in its memory, but still produce a valid checksum. In Section 5.3.4, we show that the checksum computation can have overhead to a degree, so this attack is practical. We thus emphasize that S'_d should be fully randomized by $\mathcal{V}$.

5.3.3 $\mathcal{P}_c$ Pre-compute Checksum: Challenge Buffering Attack

The attestation scheme is trivially vulnerable to challenge buffering attack that stores the challenge-response pairs directly in the memory. Upon receiving a particular challenge from $\mathcal{V}$, $\mathcal{P}_c$ looks for the corresponding checksum from its memory without computation. Since S'_d and o are received in the attestation procedure, the challenge-response stored in the memory is the tuple $\langle o, r_n \rangle$ which has the length of $l_o + l_r$. Thus, the memory can hold $m \cdot l_c/(l_o + l_r)$ records at most. Additionally, we have 2^{l_o} different receivable values. When $\mathcal{P}_c$ cannot find the record, he can choose a random response from $\{0,1\}^{l_r}$. As a consequence, the probability of computing the correct response with challenge buffering attack method for $\mathcal{P}_c$ can be expressed as follows.

$$\mathbb{P}_b(l_o, l_c, l_r, m_d, m) = b + \frac{1-b}{2^{l_r}} \text{ where } b = \frac{m \cdot l_c}{(l_o + l_r) \cdot 2^{l_o}} \tag{5.3}$$

As can be seen, $\mathbb{P}_b(l_o, l_c, l_r, m_d, m)$ is also lower-bounded by 2^{-l_r}. So we make the similar suggestion for formula (5.3) as in Section 5.3.2 that $b \cdot (1 - 2^{-l_r}) \leq 2^{-l_r}$.

5.3.4 $\mathcal{P}_c$ Forward Checksum Computation to $\mathcal{A}$: Proxy Attack

As reported in [79], the software-based attestation is particular vulnerable to the proxy attack, in which the compromised prover $\mathcal{P}_c$ forwards the challenge to the adversary $\mathcal{A}$ (a base station) and asks $\mathcal{A}$ to compute the checksum for it. In order to prevent the proxy attack, the expected checksum computation time should be no larger than a time bound, so that $\mathcal{P}_c$ does not have time to wait for the response from $\mathcal{A}$. However, one assumption should be made that $\mathcal{A}$ cannot personate $\mathcal{P}_c$ and communicate with $\mathcal{V}$ directly. Otherwise,

the software-based attestation is trivially broken. The assumption can be hold when $\mathcal{V}$ is connected to $\mathcal{P}_c$ using special channels (e.g., bus, usb) that $\mathcal{A}$ has no direct access to.

Assume the network RTT between $\mathcal{V}$ and $\mathcal{P}_g$ varies from $T_{\mathcal{V}}^{min}$ to $T_{\mathcal{V}}^{max}$ and the honest prover $\mathcal{P}_g$ can finish the checksum computation with time $d_g = n \cdot (d_{Gen} + d_{Addr} + d_{Read} + d_{Chk})$, the timing threshold d_{th} on the verifier side thus should be configured as

$$d_{th} \geq d_g + T_{\mathcal{V}}^{max} \tag{5.4}$$

to ensure the correctness of the attestation protocol defined in Section 5.2. Hence, the maximum usable time for $\mathcal{P}_c$ can be defined as $d_c(T) = d_{th} - T$, where $T \in [T_{\mathcal{V}}^{min}, T_{\mathcal{V}}^{max}]$ is the real network latency between $\mathcal{P}_c$ and $\mathcal{V}$.

On one hand, $\mathcal{P}_c$ could use $d_c(T)$ to conduct the proxy attack. If the network RTT between $\mathcal{A}$ and $\mathcal{P}_c$ varies from $T_{\mathcal{A}}^{min}$ and $T_{\mathcal{A}}^{max}$, in order to prevent the proxy attack completely, we need to make sure that $d_c(T_{\mathcal{V}}^{min}) < T_{\mathcal{A}}^{min}$, which means the proxy attack cannot be conducted even under the optimal RTT for $\mathcal{P}_c$. Thus, the attestation time for the genuine prover should be constrained by

$$d_{th} < T_{\mathcal{A}}^{min} + T_{\mathcal{V}}^{min}. \tag{5.5}$$

On the other hand, $\mathcal{P}_c$ could use $d_c(T)$ to conduct the memory recovering attack. So we calculate the δ specified in the *memory recovery attack* as follows.

$$\frac{d_{Gen} + d_{Addr} + \delta \cdot d_{Read} + d_{Chk}}{d_{Gen} + d_{Addr} + d_{Read} + d_{Chk}} = \frac{d_c(T)}{d_g} = \frac{d_{th} - T}{d_g} \tag{5.6}$$

Since, $\delta \propto d_g^{-1} \propto n^{-1}$, in order to keep the δ small, the checksum computation routine should use the largest n as possible, when formula (5.4) and (5.5) are still satisfied.

5.4 Case Studies

In this section, we analyze three well-known software-based attestation protocols, i.e., SWATT [108], SCUBA [106] and VIPER [79]. Since the generic software-based attestation scheme is configured with the parameters listed in Table 5.1, we first extract them

Parameters	SWATT	SCUBA	VIPER
l_o, l_g, l_r (bit)	2048, 16, 64	128, 16, 160	-, 32, 832
l_c, l_a (bit)	8, 14	8, 7	8, 13
m_d, m_p, m (unit)	1K, 16K, 17K	0K, 512, 58K	0K, 8K, 4120K
$T_{\mathcal{A}}^{min}, T_{\mathcal{A}}^{max}$	-	$\leq$ 22ms, 51ms	1152ns(43.34ms), 44.10ms
$T_{\mathcal{V}}^{min}, T_{\mathcal{V}}^{max}$	-	$\leq$ 22ms, 51ms	1375ns, 1375ns
d_{th}, d_g	-, 1.8s	2.915s, 2.864s	2300ns, 827ns
n, k, u	3.2E+05, 1, 8	4.0E+04, 1, 10	3, 300, 26

Table 5.2: Settings of Software-based Attestation Protocols Studied in Section 5.4

from the real protocols as shown in Table 5.2. As can be seen, our generic attestation scheme can capture existing software-based attestation protocols readily. Then, we apply the security criteria described in Section 5.3 manually to the extracted parameters to find security flaws. In the following subsections, we briefly introduce the protocols first, and then give detailed vulnerabilities and justifications grouped by the topics in **bold** font. We mark the topics with "$\star$" if they are reported for the first time in the literature.

5.4.1 SWATT

SWATT [108] randomly traverses the memory to compute the checksum. Its security is guaranteed by the side channel on time consumed in the checksum computation. SWATT does not consider network RTT, so we do not discuss time related properties for SWATT. In addition, SWATT uses RC4 as the PRNG and takes the challenge as the seed of the RC4. As the length of the challenge chosen in the SWATT is not mentioned in [108], we assume that the challenge is long enough to fully randomize the initial state of RC4, which means $l_o = 256 \cdot 8$ bits.

Unattested Data Memory. The micro-controller in SWATT has $16KB$ program memory and $1KB$ data memory. Based on the analysis of the generic attestation scheme, SWATT is insecure because it neither has Secure Erasure Phase to overwrite the data memory nor uses any additional complement to secure the data memory. In fact, the authors of

SWATT assumed in [108] that non-executable data memory can do no harm to the security of software-based attestation by mistake. In [39], Castelluccia et al. point out that the data memory should be verified in SWATT, otherwise the protocol is vulnerable to the ROP [109, 34] attack. In this work, we suggest to securely erase the data memory in SWATT by following our generic attestation scheme.

⋆**Too Large Iteration Number for Computing One Checksum.** The main loop of SWATT has only 16 assembly instructions, which takes 23 machine cycles. Inserting one if statement in the loop will cause additional 13% overhead. As a result, we assume that the recovering algorithm $\mathcal{C}$ only has time to read the memory content as $Read$ does without doing any extra computation. Hence, the success probability of the memory recovering of SWATT becomes $\mathbb{P}_m(S, \mu, \delta) = 1 - \mu$, where μ is the percentage of the modified memory. According to the formula (5.2), after setting η as suggested, we have $n \geq -64/log(1-\mu)$. When $\mu = 0.001$ which left only 16 byte memory for the adversary, we should set n as 44340, which is much smaller than the iteration number 320000 used in SWATT. In order to increase the difficulty of attacking the attestation protocol and traverse the memory address in the platform, more rounds of checksum computation could be conducted. According to formula (5.1), when $\mu = 0.001$, $n = 44340$ and $c = 2$ (the same setting in SWATT), we have $k \geq 11$. So we should conduct the checksum computation for 11 times. By using this new configuration, the overall memory access time is approximately the same as SWATT while security guarantee becomes dramatically better.

5.4.2 SCUBA

SCUBA [106] is a software-based attestation protocol that based on Indisputable Code Execution (ICE). Rather than attesting the whole memory, the ICE offers security guarantee by only verifying a small portion of the code. The $Read$ and Chk implemented in the ICE scheme are different from those given in Section 5.2.2. However, they can be generalized into our framework. In SCUBA, $Read$ not only reads the memory content, but

also returns the Program Pointer (PC), the current address, the current generator, the loop counter and other registers. The Chk function then computes the checksum based on all of them. In order to compute the correct checksum for the modified attestation routine, the malicious prover has to simulate the execution for all of them, which thus lead to large and detectable overhead on the computation time. If the malicious prover do not change the attested code, the attested code can update the prover's whole memory to a genuine state so the malicious code shall be removed from the prover.

⋆**Proxy Attack is Indefensible.** In SCUBA, network RTT is explicitly evaluated in the experiment as summarized in Table 5.2. The prover in SCUBA communicates with the verifier over wireless network. Even though the adversary is assumed to be physically absent during the attestation in SCUBA, this assumption seems to be too strong to be hold when a wireless network presents. Thus, we give a detailed analysis for the proxy attack to SCUBA as follows.

According to [106], the maximum network RTT is $51ms$ in SCUBA. By observing the experiment results, the minimum network RTT should be no larger than $22ms$. As the adversary and the verifier share the same wireless network, the network latency for their communication with the prover should be indifferent. So we have $T_{\mathcal{A}}^{min} = T_{\mathcal{V}}^{min} \leq 22ms$ and $T_{\mathcal{A}}^{max} = T_{\mathcal{V}}^{max} = 51ms$. According to formula (5.4), we have $d_{th} \geq d_g + T_{\mathcal{V}}^{max} \geq 51ms$. On the other hand, according to formula (5.5), we have $d_{th} < T_{\mathcal{A}}^{min} + T_{\mathcal{V}}^{min} \leq 44ms$. Hence, we cannot find a valid threshold d_{th} from this network configuration. When the adversary presents in the attestation, the proxy attack thus cannot be defended by SCUBA without additional assumptions.

Moreover, if the verifier does not communicate with the prover with a secure channel (e.g., the verifier uses the wireless network to the communicate with the prover in this case), the adversary can personate the prover and send the checksum to the verifier directly. Since the adversary can compromise the prover, he can obtain the secret key stored in

the prover as well. So encrypting the wireless channel will not work. We suggest that the verifier should communicate with the prover in an exclusive method, such as the usb connection, which is also inexpensive. More importantly, the adversary cannot use this communication method as it is highly controllable.

Security Claim Justification. Our framework can not only be used to find potential vulnerabilities, but also give justifications to the security claims made in existing works. In SCUBA [106], the malicious prover may exploit the network latency to conduct memory recovering attack without being detected. However, if the timing overhead of the attack is even larger than the largest network latency, the attack then becomes detectable. According to this, the authors of SCUBA claim that the checksum computation time adopted in SCUBA can always detect the memory copy attack, which is the most efficient memory recovering attack method known to the authors, even if the malicious prover can communicate without network delay.

In this work, we can justify their security claim with our framework. When the proxy attack is not considered in SCUBA, increasing the checksum computation time does not introduce vulnerability. According to formula (5.6), we have $d_c(T)/d_g = (d_{th} - T)/d_g$. The experiment results in [106] show that the memory copy attack is most efficient attack which introduces 3% overhead to the checksum computation. In order to detect the memory copy attack, we should ensure that $\forall T \in [T_{\mathcal{V}}^{min}, T_{\mathcal{V}}^{max}], d_c(T)/d_g < 1.03$. As we assume that the malicious prover can communicate without network delay, we set $T_{\mathcal{V}}^{min}$ as 0. By applying formula (5.4), we have $d_g > 1700ms$. Since d_g chosen in SCUBA is indeed larger than $1700ms$, the security claim made by the authors is valid.

5.4.3 VIPER

VIPER [79] is a software-based attestation scheme designed to verify the integrity of peripherals' firmware in a typical x86 computer system. They are proposed to defend all known software-based attacks, including the proxy attack.

⋆**Absence of Random Function.** VIPER uses a similar design as ICE scheme, while its generators are not produced by a PRNG during the checksum computation, which does not comply to our generic attestation scheme. The authors implement the checksum function into 32 code blocks. One register is updated in every code block with the memory content and the program counter (PC). Both of the code block and the memory address are chosen based on the current checksum. Thus, the randomness of the checksum is purely introduced by the PC and the memory content. However, the PC is incremented in a deterministic way inside each code block and the memory content usually is biased as illustrated in Section 5.3.2. As the randomness could be biased, the adversary can traverse all challenge values and he may find some memory addresses that are unreachable for the checksum computation routine, as we discussed in Section 5.3.1. Hence, the security provided by VIPER is unclear.

⋆**Insufficient Iteration Number.** In VIPER, the number of iterations used in the checksum computation routine is only 3, which leads to at least 23 unused registers in the attestation. Vulnerabilities may be introduced as discussed in Section 5.3.1. Even if the registers are chosen in a fully randomized manner and the adversary cannot predict which register will be used beforehand, the malicious prover still has a high probability to use some registers without being detected. In fact, two or even one register could be enough for conducting an attack in practice.

5.5 Related Works

A large amount of software-based attestation protocols have been designed and implemented [66, 108, 63, 110, 107, 106, 120, 62, 7, 97, 79, 71]. Specifically, SWATT [108] is a software-based attestation scheme that uses the response timing of the memory checksum computation to identify the compromised embedded devices. In order to prevent replay attack, the prover's memory is traversed in SWATT in a random manner based on a challenge

sent from the verifier. Rather than attesting the whole memory content, SCUBA [106] only checks the protocol implemented in the embedded devices and securely updates the memory content of the embedded devices after the attestation is finished successfully. It is based on the ICE (Indisputable Code Execution) checksum computation scheme, which enables the verifier to obtain an indisputable guarantee that the SCUBA protocol will be executed as untampered in the embedded devices. VIPER [79] is later proposed to defense against the adversary who can communicate with the embedded devices during the attestation. Network latency is consider in VIPER to prevent the proxy attack. Perito et al. [97] develop a software-based secure code update protocol. It first overwrites the target device's whole memory with random noise and then asks the target device to generate a checksum based on its memory state. The target device could generate the correct checksum only if it has erased all its memory content, so the malicious code should also be removed. Besides the attestation protocol designed for resource constrained devices, Seshadri et al. [107] develop the software-based attestation protocol named Pioneer for the Intel Pentium IV Xeon Processor with x86 architecture.

However, the software-based attestation protocol design is challenging and error-prone [111, 39]. Hence, it becomes necessary and urgent to develop an analysis framework for the attestation protocol design. Armknecht et al. [15] recently provide a security framework for the analysis and design of software attestation. In their work, they assume the cryptographic primitives such as Pseudo-Random Number Generators (PRNGs) and hash functions might be insecure and give a upper-bound to the advantage of the malicious prover in the attestation scheme. They mainly consider six factors: (1) the memory content could be biased; (2) the memory addresses traversed in the checksum computation may not be fully randomized; (3) the memory addresses could be computed without using the default method; (4) the correct checksum could be computed without finishing the checksum computation routine; (5) the checksum could be generated without using the default checksum computation function; (6) the challenge-response pairs could be pre-computed and stored

in the memory. In this work, we do not consider factor (2-5) based on two reasons. First, the attestation routine used in the protocol can be updated at runtime, so we can always update the cryptographic functions to meet the higher security standard and requirement. For instance, since the hash function like *MD5* could be insecure, we can replace it with *SHA-2* or *SHA-3* to reclaim security. More importantly, the upper-bounds of the factor (2-5) are very hard to measure in practice. For example, given a well-known weak hash function like *MD5*, it is hard to measure the *time-bounded pseudo-randomness*, corresponding to factor (2), defined in [15]. Comparing with [15], we additionally consider observable network latency, stronger threat model, unpredictable data memory, several security criteria and various attack schemes. More importantly, our framework has been successfully applied to several existing software-based attestation protocols to find vulnerabilities.

5.6 Discussions

In this work, we present a practical analysis framework for software-based attestation scheme. We explicitly consider the network latency and the data memory in the system. Furthermore, the adversary presented in this work can not only reprogram the compromised provers before the attestation but also communicate with them during the attestation. We successfully apply our framework to three well-known software-based attestation protocols manually. The results show that our framework can practically find security flaws in their protocol design and give justifications to their security claims.

The deployment environment, including device architecture, network environment, efficiency requirement, etc. usually complicates the correctness of the software-based attestation protocols. Specifically, identifying the most effective overwriting and recovering algorithms becomes very hard, which limits the application of our framework. For future works, we believe that fine-grain measurement for the overwriting and recovering algorithms in the practical application context is useful. Another future work is investi-

gating the impact of timing requirement when the attestation efficiency is concerned. In this work, we assume that software-based attestation can take as much time as it needs. Nevertheless, in reality, we may require the attestation protocols to be finished within a timing threshold. Hence, the probability of identifying the compromised prover will be affected, and choosing the right configurations (e.g., the iteration number) then becomes more challenging.

Chapter 6

Conclusions

In the thesis, we first present an analysis to a vehicle charging protocol in Chapter 2 that considers many security properties including secrecy, authentication and privacy. During the analysis, we find several weaknesses of the existing tools as they either make strong abstractions during the verification or cannot verify security protocols for an unbounded number of sessions. Hence, we propose a verification framework that can verify protocols of an unbounded number of sessions without abstraction, which is particularly useful for verifying timed security protocols as shown in Chapter 3. We prove the partial correctness of the verification algorithm and use it to check many security protocols efficiently. Considering the timing is fixed in Chapter 3 but it should be flexible in design, we extend our framework with capabilities to verify parameterized timed protocols in Chapter 4. Furthermore, we develop the *timed applied π-calculus* as a specification language so that timed security protocols can be specified in a concise and natural way. However, the security protocols may consider physical properties in their execution context so that they cannot be specified and verified using symbolic verification method directly. In order to analyze the protocols with physical properties, we propose an analysis approach to generalize the

protocol design into a generic scheme and check the correctness based on the scheme in Chapter 5. We use this approach to analyze a family of the software-based attestation protocols and find several security weaknesses in them.

Based on the above works, I believe that our framework can be applied to verifying real security protocol specifications and implementations efficiently and automatically. Extensions required for specific domains can be extended to our framework readily. By using my current works as a verification foundation, I would like to continue my research in the following research topics.

Automatic verification for security protocol implementations. The protocol implementation usually does not completely comply with its formal specification. This can result from the incomplete interface specifications, additional environment requirements, etc. In order to ensure the correctness of protocol implementation, studying the approach that extracts the security protocol directly from their implementations and verifies it in our framework can be very interesting. The basic idea is as follows. First, we need to translate the implementation into an intermediate representation consisting of branches guarded by condition checking, API invoking and network communication using Control Flow Graph (CFG) [8]. Then, the intermediate model can be transformed into our verification framework based on the functional mapping between the APIs and their symbolic representations. Finally, if any security flaw is found during the verification, we need to validate it in the original implementation and refine the protocol abstraction whenever false alarm occurs. I believe the automatic verification of security protocol implementations is extremely promising.

A heuristic method for pruning non-terminable verification branches. The security protocol verification for an unbounded number of sessions has been proved as undecidable, so the termination of verification cannot be guaranteed in general. In our framework, the nontermination is introduced by two factors: the infinite knowledge deduction and the

infinite timing expansion. We resolve the second one by introducing over-approximation to the timing constraints. However, the first factor still persists in our framework. Even though no general approach exists for deciding the termination of verification process, detecting the non-terminable cases heuristically is still possible so that we can prune some of the verification branches without affecting the final result. This work could help us to verify large-sized and complex protocols that we cannot verify currently, because our verification algorithm only considers the general approach at present.

A compositional approach for automatic security protocol generation. Designing security protocols is challenging and error-prone. Fortunately, automatic generation of security protocols is possible. Several methods [99, 98] have been previously proposed by many researchers. However, existing methods are inefficient. Recently, a new compositional security paradigm is proposed, which is called universal composable (UC) security [37]. It has a salient property that a secure protocol can be constructed with an arbitrary set of protocol components compositionally. Under this paradigm, we can specify these security protocol components in our security protocol verification framework, and efficiently search for security protocols with the required properties under our framework.

Extending PAT to verify security protocols. PAT [117] has been successfully applied to verifying security protocols [88]. PAT is a compact tool which supports verification of real-time systems [10, 116] with model checking [114, 81] as well as hybrid approach [115]. I believe that extending PAT with security protocol verification could make it a comprehensive verification tool for the end-users. The security protocol verification could also benefit from its explicit model checking engine to handle protocol states and global variables.

Bibliography

[1] Models of the electric vehicle charging protocol in ProVerif and Tamarin. `http://www.comp.nus.edu.sg/~li-li/r/saevcp.html`.

[2] Timed and untimed models for TAuth. `http://www.comp.nus.edu.sg/~li-li/r/time.html`.

[3] Trusted Platform Module. `http://www.trustedcomputinggroup.org/developers/trusted_platform_module`.

[4] M. Abadi. Explicit communication revisited: Two new attacks on authentication protocols. *IEEE Trans. Software Eng.*, 23(3):185–186, 1997.

[5] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.

[6] M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.

[7] T. AbuHmed, N. Nyamaa, and D. Nyang. Software-based remote code attestation in wireless sensor network. In *Proc. of the Global Communications Conference (GLOBECOM'09)*, pages 1–8. IEEE, 2009.

[8] F. E. Allen. Control flow analysis. In *Proc. of a Symposium on Compiler Optimization*, pages 1–19. ACM, 1970.

[9] R. Anderson and R. Needham. Programming satan's computer. In *Computer Science Today*, volume 1000, pages 426–440. Springer, 1995.

[10] É. André, Y. Liu, J. Sun, and J. S. Dong. Parameter synthesis for hierarchical concurrent real-time systems. *Real-Time Systems*, 50(5-6):620–679, 2014.

[11] H. Andréka, I. Németi, and J. van Benthem. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[12] M. Arapinis, T. Chothia, E. Ritter, and M. Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF)*, pages 107–121. IEEE CS, 2010.

[13] M. Arapinis, E. Ritter, and M. D. Ryan. StatVerif: Verification of stateful processes. In *Proc. 24th IEEE Computer Security Foundations Symposium (CSF)*, pages 33–47. IEEE CS, 2011.

[14] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proc. of 1997 IEEE Symposium on Security and Privacy (S&P'97)*, pages 65–71. IEEE CS, 1997.

[15] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. A security framework for the analysis and design of software attestation. In *Proc. of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, pages 1–12. ACM, 2013.

[16] M. H. Au, W. Susilo, and Y. Mu. Constant-size dynamic *k*-TAA. In *Proc. 5th Conference on Security and Cryptography for Networks (SCN)*, volume 4116 of *LNCS*, pages 111–125, 2006.

[17] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *SAS*, pages 213–229. Springer, 2002.

[18] G. Bai. *Formally Analyzing and Verifying Secure System Design and Implementation*. PhD thesis, National University of Singapore, 2015.

[19] G. Bai, J. Hao, J. Wu, Y. Liu, Z. Liang, and A. Martin. TrustFound: Towards a Formal Foundation for Model Checking Trusted Computing Platforms. In *Proc. of the 19th International Symposium on Formal Methods (FM)*, pages 110–126, 2014.

[20] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AuthScan: Automatic Extraction of Web Authentication Protocols from Implementations. In *Proc. of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, February 2013.

[21] G. Bai, J. Sun, J. Wu, Q. Ye, L. Li, J. S. Dong, and S. Guo. All Your Sessions are Belong to us: Investigating Authenticator Leakage through Backup Channels on Android. In *Proceedings of the 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2015.

[22] E. B. Barker, W. C. Barker, W. E. Burr, W. T. Polk, and M. E. Smid. SP 800-57. Recommendation for key management. Technical report, National Institute of Standards & Technology, 2007.

[23] D. A. Basin, S. Capkun, P. Schaller, and B. Schmidt. Formal reasoning about physical properties of security protocols. *ACM Trans. Inf. Syst. Secur.*, 14(2):16, 2011.

[24] D. A. Basin and C. J. F. Cremers. From dolev-yao to strong adaptive corruption: Analyzing security in the presence of compromising adversaries. *IACR Cryptology ePrint Archive*, 2009:79, 2009.

[25] D. A. Basin and C. J. F. Cremers. Degrees of security: Protocol guarantees in the face of compromising adversaries. In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010.

[26] D. A. Basin and C. J. F. Cremers. Modeling and analyzing security in the presence of compromising adversaries. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, volume 6345 of *Lecture Notes in Computer Science*, pages 340–356. Springer, 2010.

[27] G. Bella and L. C. Paulson. Kerberos version 4: Inductive analysis of the secrecy goals. In *ESORICS*, pages 361–375. Springer, 1998.

[28] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[29] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW*, pages 82–96. IEEE CS, 2001.

[30] B. Blanchet. From secrecy to authenticity in security protocols. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 342–359. Springer, 2002.

[31] B. Blanchet. Automatic proof of strong secrecy for security protocols. In *Proc./ of IEEE Symposium on Security and Privacy*, pages 86–. IEEE Computer Society, 2004.

[32] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebr. Program.*, 75(1):3–51, 2008.

[33] S. Brands and D. Chaum. Distance-bounding protocols (extended abstract). In *EUROCRYPT*, volume 765 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 1993.

[34] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proc. of the 2008 ACM Conference on Computer and Communications Security (CCS'08)*, pages 27–38. ACM, 2008.

[35] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[36] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal analysis of kerberos 5. *Theor. Comput. Sci.*, 367:57–87, 2006.

[37] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. of the 42nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.

[38] S. Capkun and J.-P. Hubaux. Secure positioning in wireless networks. *IEEE Journal on Selected Areas in Communications*, 24(2):221–232, 2006.

[39] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proc. of the 2009 ACM Conference on Computer and Communications Security (CCS'09)*, pages 400–409. ACM, 2009.

[40] CCITT. The directory authentication framework - Version 7, 1987. Draft Recommendation X.509.

[41] I. Cervesato, N. A. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *CSFW*, pages 55–69. IEEE CS, 1999.

[42] I. Cervesato, N. A. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proc. 12th IEEE Computer Security Foundations Workshop (CSFW)*, pages 55–69. IEEE CS, 1999.

[43] V. Cheval and B. Blanchet. Proving more observational equivalences with proverif. In *Principles of Security and Trust - Second International Conference, POST 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7796 of *Lecture Notes in Computer Science*, pages 226–246. Springer, 2013.

[44] Y.-G. Choi, J. Kang, and D. Nyang. Proactive code verification protocol in wireless sensor network. In *Computational Science and Its Applications (ICCSA'07)*, volume 4706 of *LNCS*, pages 1085–1096. Springer, 2007.

[45] T. Chothia, B. Smyth, and C. Staite. Automatically checking commitment protocols in proverif without false attacks. In *POST*, pages 137–155, 2015.

[46] A. Church. *A function of positive integers is effectively calculable only if recursive*. 1936.

[47] R. Corin, S. Etalle, P. H. Hartel, and A. Mader. Timed model checking of security protocols. In *FMSE*, pages 23–32. ACM, 2004.

[48] C. Cremers. The Scyther tool: Verification, falsification, and analysis of security protocols. In *CAV*, pages 414–418. Springer, 2008.

[49] C. J. F. Cremers. Formally and practically relating the ck, ck-hmqv, and eck security models for authenticated key exchange. *IACR Cryptology ePrint Archive*, 2009:253, 2009.

[50] C. J. F. Cremers. Session-state reveal is stronger than ephemeral key reveal: Attacking the naxos authenticated key exchange protocol. In *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*, volume 5536 of *Lecture Notes in Computer Science*, pages 20–33, 2009.

[51] C. J. F. Cremers and M. Feltz. Beyond eck: Perfect forward secrecy under actor compromise and ephemeral-key reveal. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, volume 7459 of *Lecture Notes in Computer Science*, pages 734–751. Springer, 2012.

[52] C. J. F. Cremers, S. Mauw, and E. P. de Vink. Injective synchronisation: An extension of the authentication hierarchy. *Theor. Comput. Sci.*, 367(1-2):139–161, 2006.

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

[54] G. Delzanno and P. Ganty. Automatic verification of time sensitive cryptographic protocols. In *TACAS*, pages 342–356. Springer, 2004.

[55] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, 1981.

[56] D. Dolev and A. C.-C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[57] N. Dong, H. L. Jonker, and J. Pang. Formal analysis of privacy in an eHealth protocol. In *Proc. 17th European Symposium on Research in Computer Security (ESORICS)*, volume 7459 of *LNCS*, pages 325–342. Springer, 2012.

[58] P. England, B. W. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Computer*, 36(7):55–62, 2003.

[59] N. Evans and S. Schneider. Analysing time dependent security properties in csp using pvs. In *ESORICS*, pages 222–237. Springer, 2000.

[60] F. J. T. Fábrega. Strand spaces: proving security protocols correct. *Journal Computer Security*, 7(2-3):191–230, 1999.

[61] A. Francillon, B. Danev, and S. Capkun. Relay attacks on passive keyless entry and start systems in modern cars. In *NDSS*. The Internet Society, 2011.

[62] R. W. Gardner, S. Garera, and A. D. Rubin. Detecting code alteration by creating a temporary memory bottleneck. *IEEE Transactions on Information Forensics and Security*, 4(4), 2009.

[63] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Proc. of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 23–32. IEEE CS, 2005.

[64] J. Hao, Y. Liu, W. Cai, G. Bai, and J. Sun. vTRUST: A Formal Modeling and Verification Framework for Virtualization Systems. In *15th International Conference on Formal Engineering Methods (ICFEM)*, pages 329–346, 2013.

[65] G. Jakubowska and W. Penczek. Is your security protocol on time? In *FSEN*, pages 65–80. Springer, 2007.

[66] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proc. of the 12th Conference on USENIX Security Symposium*, pages 21–21. USENIX, 2003.

[67] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Proc. of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09)*, pages 115–124. IEEE, 2009.

[68] A. Klimov and A. Shamir. New cryptographic primitives based on multiword t-functions. In *Proc. of the 11th International Workshop on Fast Software Encryption (FSE'04)*, volume 3017 of *LNCS*, pages 1–15. Springer, 2004.

[69] J. Kohl and B. C. Neuman. *The Kerberos Network Authentication Service (Version 5). Internet Request for Comments RFC-1510*. RFC Editor, 1993.

[70] J. T. Kohl, B. C. Neuman, and T. Y. T'so. The evolution of the kerberos authentication system. In *Distributed Open Systems*, pages 78–94. IEEE CS, 1994.

[71] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *Proc. of IEEE Symposium on Security and Privacy (S&P'12)*, pages 239–253. IEEE CS, 2012.

[72] H. Krawczyk. Skeme: a versatile secure key exchange mechanism for internet. In *NDSS*, pages 114–127. IEEE CS, 1996.

[73] S. Kremer and R. Künnemann. Automated analysis of security protocols with global state. In *Proceedings of the Symposium on Security and Privacy*, pages 163–178, 2014.

[74] LDAP Account Manager. Kerberos V implementation heimdal-1.5.2. `http://www.h5l.org`, 2014.

[75] L. Li, H. Hu, J. Sun, Y. Liu, and J. S. Dong. Practical analysis framework for software-based attestation scheme. In *Proc. 16th International Conference on Formal Engineering Methods*, pages 284–299. Springer, 2014.

[76] L. Li, J. Pang, Y. Liu, J. Sun, and J. S. Dong. Symbolic analysis of an electric vehicle charging protocol. In *Proc. 19th International Conference on Engineering of Complex Computer Systems*, pages 11–18. Springer, 2014.

[77] L. Li, J. Sun, Y. Liu, and J. S. Dong. Tauth: Verifying timed security protocols. In *Proc. 16th International Conference on Formal Engineering Methods*, pages 300–315. Springer, 2014.

[78] L. Li, J. Sun, Y. Liu, and J. S. Dong. Verifying parameterized timed security protocols. In *Proc. of the 20th International Symposium on Formal Methods*, page 342–359. Springer, 2015.

[79] Y. Li, J. M. McCune, and A. Perrig. Viper: verifying the integrity of peripherals' firmware. In *Proc. of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, pages 3–16. ACM, 2011.

[80] J. K. Liu, M. H. Au, W. Susilo, and J. Zhou. Enhancing location privacy for electric vehicles (at the right time). In *Proc. 17th European Symposium on Research in*

Computer Security (ESORICS), volume 7459 of *LNCS*, pages 397–414. Springer, 2012.

[81] Y. Liu, J. Sun, and J. S. Dong. An analyzer for extended compositional process algebras. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 919–920, 2008.

[82] G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[83] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *TACAS*, pages 147–166. Springer, 1996.

[84] G. Lowe. A family of attacks upon authentication protocols. Technical report, Department of Mathematics and Computer Science, University of Leicester, 1997.

[85] G. Lowe. A hierarchy of authentication specification. In *CSFW*, pages 31–44. IEEE Computer Society, 1997.

[86] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[87] G. Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(1):89–146, 1999.

[88] A. T. Luu, J. Sun, Y. Liu, J. S. Dong, X. Li, and T. T. Quan. Seve: automatic tool for verification of security protocols. *Frontiers of Computer Science in China*, 6(1):57–75, 2012.

[89] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The Tamarin prover for the symbolic analysis of security protocols. In *CAV*, pages 696–701. Springer, 2013.

[90] MIT. Kerberos V implementation krb5-1.13. `http://web.mit.edu/kerberos/`, 2014.

[91] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ;. In *S&P*, pages 141–151, 1997.

[92] M. Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, 4(2):151–158, 1991.

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

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

[95] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. *The Kerberos Network Authentication Service (Version 5). RFC-4120*. RFC Editor, 2005.

[96] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proc. 11th Annual International Cryptology Conference (CRYPTO)*, volume 576 of *LNCS*, pages 129–140. Springer, 1991.

[97] D. Perito and G. Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *Proc. 15th European Symposium on Research in Computer Security (ESORICS'10)*, volume 6345 of *LNCS*, pages 643–662. Springer, 2010.

[98] A. Perrig and D. Song. Looking for diamonds in the desert - extending automatic protocol generation to three-party authentication and key agreement protocols. In *Proc. of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 64–76, 2000.

[99] A. Perrig and D. X. Song. A first step towards the automatic generation of security protocols. In *Proc. of the Network and Distributed System Security Symposium (NDSS), San Diego, California, USA*, 2000.

[100] K. B. Rasmussen, C. Castelluccia, T. S. Heydt-Benjamin, and S. Capkun. Proximity-based access control for implantable medical devices. In *CCS*, pages 410–419. ACM, 2009.

[101] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(1):147–190, 1999.

[102] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proc. of the 13th USENIX Security Symposium*, pages 223–238. USENIX, 2004.

[103] N. Sastry, U. Shankar, and D. Wagner. Secure verification of location claims. In *Workshop on Wireless Security*, pages 1–10. ACM, 2003.

[104] B. Schmidt, S. Meier, C. J. F. Cremers, and D. 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. IEEE, 2012.

[105] S. Sedighpour, S. Capkun, S. Ganeriwal, and M. B. Srivastava. Implementation of attacks on ultrasonic ranging systems (demo). In *SenSys*, page 312. ACM, 2005.

[106] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. K. Khosla. Scuba: Secure code update by attestation in sensor networks. In *Proc. of the 2006 ACM Workshop on Wireless Security*, pages 85–94. ACM, 2006.

[107] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 1–16. ACM, 2005.

[108] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. Swatt: Software-based attestation for embedded devices. In *Proc. of the 2004 IEEE Symposium on Security and Privacy (S&P'04)*, pages 272–. IEEE CS, 2004.

[109] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proc. of the 2007 ACM Conference on Computer and Communications Security (CCS'07)*, pages 552–561. ACM, 2007.

[110] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *Proc. of the 2nd European Workshop of Security and Privacy in Ad-hoc and Sensor Networks (ESAS'05)*, pages 27–41. Springer, 2005.

[111] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proc. of the 13th USENIX Security Symposium*, pages 89–102. USENIX, 2004.

[112] D. X. Song. Athena: A new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop, CSFW 1999, Mordano, Italy, June 28-30, 1999*, pages 192–202. IEEE Computer Society, 1999.

[113] D. X. Song, S. Berezin, and A. Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1-2):47–74, 2001.

[114] J. Sun, Y. Liu, and J. S. Dong. Model checking CSP revisited: Introducing a process analysis toolkit. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings*, pages 307–322, 2008.

[115] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating specification and programs for system modeling and verification. In *TASE 2009, Third IEEE International Symposium on Theoretical Aspects of Software Engineering, 29-31 July 2009, Tianjin, China*, pages 127–135, 2009.

[116] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and É. André. Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Trans. Softw. Eng. Methodol.*, 22(1):3, 2013.

[117] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: towards flexible verification under fairness. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 709–714, 2009.

[118] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings*, pages 160–171. IEEE Computer Society, 1998.

[119] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 1999.

[120] Y. Yang, X. Wang, S. Zhu, and G. Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *Proc. 26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, pages 219–230. IEEE CS, 2007.