A Formal Specification and Verification Framework for Timed Security Protocols

Li Li, Jun Sun, Yang Liu, Meng Sun, and Jin Song Dong

Abstract—Nowadays, protocols often use time to provide better security. For instance, lifetime of critical credentials and latency of physical networks are often introduced as safety thresholds in system designs. However, using time correctly in protocol design is challenging, due to a lack of time related formal specification and verification techniques. Thus, we propose a comprehensive analysis framework to formally specify as well as automatically verify timed security protocols. A parameterized method is introduced in our framework to handle flexible timing constants whose values cannot be decided in the protocol design stage. In this work, we first propose *timed applied π-calculus* as a formal specification language for timed protocols. It can express computation of continuous time as well as application of cryptographic functions. Then, we define its formal semantics based on *timed logic rules*, which facilitates efficient verification against various authentication and secrecy properties. Given a parameterized security protocol, the verification result either produces secure configuration methods of its parameters, or reports an attack that works for any parameter values. The correctness of our verification algorithm has been formally proved. We evaluate our framework with multiple timed and untimed security protocols and successfully find a previously unknown timing attack in Kerberos V.

Index Terms—Timed Security Protocol, Timed Applied π-calculus, Parameterized Verification, Secrecy and Authentication

1 Introduction

Time is a double edged sword for security protocols. On one hand, time, as a globally shared measurement, provides a simple way to synchronize and coordinate multiple processes. Thus, it is used in many security protocols as a powerful tool. For instance, distance bounding protocols [3], [4], [5] use transmission time to measure the distance between protocol participants; interactive protocols [6], [7] limit the lifetime of messages to achieve better security. In fact, timeout is used almost universally in practice. On the other hand, time also introduces a range of attack surfaces. For instance, a security protocol, whose correctness heavily relies on time, could be broken if the expected timing coordination is compromised; given a session key with limited lifetime, the adversary might be able to extend its lifetime without proper authorization [8]. As a consequence, we believe that verification of timed security protocols is an important research problem.

Specifically, the time related security of distributed processes relies on various timing constraints, which are designed based on the knowledge of the protocol execution context. For instance, in a message transmission protocol, the message receiver can check the message freshness using a timing constraint $t' - t \leq p_m$, where t' is the message receiving time, t is the message generation time and p_m is the maximum message lifetime. More importantly, the message lifetime p_m should be configured based on the knowledge of the minimal network latency p_n, among other things. That is, some correlation between p_m and p_n must be satisfied. In practice, knowing a concrete value of the network latency at the protocol design stage is highly non-trivial. Thus, it is desirable if one could leave p_m and p_n as *parameters* (symbols with fixed but unknown values) and automatically obtain their secure *configurations* (timing constraints) that ensure protocol security. The benefit is obvious: having the secure configurations of the above example, for any particular value of the network latency p_n observed in practice, the users can easily select a secure (and perhaps more efficient, e.g., in reducing system execution time) value for the message lifetime p_m. Comparing with the *standard verification problem* where the values of the parameters are given, computing the secure configurations is more complicated as it boils down to verify timed security protocols for any valuation of the parameters. It is often known as the *parameterized verification problem*.

In view of the above research problems, in this work, we develop a self-contained framework, which facilitates not only formal specification but also automatic verification of the timed security protocols. It can solve the above-mentioned *standard verification problem* as well as the *parameterized verification problem*. It is highly non-trivial because of the following technical challenges. (1) To model the timed security protocols naturally, we need to develop a high-level specification language with time related operations and measurements. On the other hand, in order to facilitate an efficient verification method, it must have a concise

This work is a substantial extension to [1], [2] with the following extra contents. First, we develop an intuitive protocol specification language for timed security protocols. Then, the semantics of our specification language is formally defined using timed logic rules proposed in [1], [2]. Third, in addition to check the secrecy [2] and non-injective authentication [1] properties, the verification method for injective authentication properties is developed in this work. Fourth, several case studies are added in this work, e.g., the injective version of Wide Mouthed Frog protocol and the commitment protocols.

Li Li and Jun Sun are with ISTD, Singapore University of Technology and Design, Singapore. E-mail: {sunjun, li_li}@sutd.edu.sg.
Yang Liu is with School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: yangliu@ntu.edu.sg
Meng Sun is with School of Mathematical Sciences, Peking University, China. E-mail: sunmeng@math.pku.edu.cn
Jin Song Dong is with School of Computing, National University of Singapore, Singapore. E-mail: yangliu@ntu.edu.sg
The corresponding author Li Li is with ISTD, Singapore University of Technology and Design, Singapore. E-mail: li_li@sutd.edu.sg. Phone: +6590141554.

and compact low-level semantics. In this way, the timed security protocol thus can be naturally specified as well as efficiently verified. (2) In the context of vulnerable network such as Internet, where communications are exposed to the adversary, we need to capture the capability of the adversary precisely so as to prove the security properties, e.g., the critical information cannot be leaked and the protocol works as intended. (3) 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. (4) A protocol design might contain multiple timing parameters, e.g., the network latency and the session key lifetime, which could affect the its security. Hence, calculating the secure relation among these parameters automatically is far more challenging than verifying the protocol with fixed parameter values.

Contributions. In this work, we first propose a *timed applied π-calculus* to specify timed security protocols, which extends *applied π-calculus* [9] with time related operations and measurements. As shown in Section 2, *timed applied π-calculus* can be used to model timed and untimed security protocols in a natural manner. In particular, symbolic parameters can be specified in the timing constraints. As a result, the protocol correctness can be verified by generating secure configurations on these parameters automatically if possible. Otherwise, an attack shall be identified for arbitrary parameter values. Additionally, secrecy property, non-injective authentication property and injective authentication property can be formally specified in our framework as shown in Section 3.

Given the *timed applied π-calculus*, we define its semantics based on *timed logic rules* in Section 4. The adversary then can be precisely captured by the *timed logic rules*, which are originally introduced in [1], [2]. Since all of the rules can be used for an infinite number of times during the verification, the protocols are verified for an unbounded number of protocol sessions.

Given the *timed logic rules*, we develop our verification algorithms against different security properties in Section 5. The verification result either proves the correctness of the protocol by providing secure configurations of the parameters, or claiming an attack that works for any parameter configuration. We prove that our verification algorithms always produce correct results.

Finally, we implement our method into a tool named Security Protocol Analyzer (SPA). In order to handle the parameters in the timing constraints, we utilize the Parma Polyhedra Library (PPL) [10] in our tool to represent and to manipulate the timing constraints. We evaluate our approach with several security protocols in Section 6. We have found a time related attack successfully using SPA in the official document of Kerberos V [11].

Structure of the paper. In Section 2, we develop the *timed applied π-calculus* as a specification language for modeling timed security protocols. We illustrate the Wide Mouthed Frog (WMF) [6] as a running example in the following paper. In Section 3, we formally defined authentication and secrecy properties based on the events and processes introduced in the *timed applied π-calculus*. In Section 4, we introduce the *timed logic rule* [1], [2], and use it to define the semantics of the *timed applied π-calculus*. The verification algorithms are given in Section 5. We prove that our algorithms always give correct results when the verification terminates. The experiment results are shown in Section 6, where a new attack of Kerberos V is found in RFC 4120 [11]. The related works are described in Section 7. Finally, we draw conclusions in Section 8.

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

a. The expression with the brackets '$[E]$' means that E can be omitted.

TABLE 1
Syntax of Timed Applied π-Calculus

2 TIMED APPLIED π-CALCULUS

In this section, we propose *timed applied π-calculus* as a specification language for timed protocols. It extends the *applied π-calculus* [9] with timing related operations and measurements. We use the Wide Mouthed Frog protocol [6] as a running example to demonstrate the language features.

2.1 Syntax

Comparing with the syntax of the *applied π-calculus*, generating, checking and encoding timestamps are allowed in the *timed applied π-calculus*. The syntax of the *timed applied π-calculus* is shown in Table 1, which consists of five expression categories, i.e., *messages*, *parameters*, *constraints*, *configurations* and *processes*. The new structures and expressions are highlighted with the **bold** font in Table 1.

Generally, messages represent the data transmitted in the process. They can be composed from *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 numbers in the processes; *timestamps* are clock readings extracted during the process execution; and *variables* are memory spaces for holding *messages*. Additionally, *parameters* are pre-configured constants (e.g., p_m) and persistent environment settings (e.g., p_n) in the protocol.

Functions can be generally 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. When m is exactly the same as $f(m_1, m_2, \ldots, m_n)$, we call the function as *constructor*; otherwise, it is a *destructor*. 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) for constructors, we omit '$\Rightarrow m$' in the definition. 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 a non-negative amount of time. We can simply write its definition as $enc_s(m, k)$. Similarly, the symmetric decryption function dec_s can be defined as $dec_s(enc_s(m, k), k) \Rightarrow m$ where m and k have the same meaning as above. For illustration purpose, some frequently used functions are presented in Table 2. Notice that the input and output messages in the function definition can only be constructors or variables.

The constraint set $B = \mathcal{CS}(t_1, t_2, \ldots, t_n, p_1, p_2, \ldots, p_m)$ represents a set of linear constraints over timestamps and parameters, which can acts as guard conditions and timing assumptions in the protocol. Generally, when $\sim \in \{<, \leq\}$ and for any $i \in \{1 \ldots n\}, j \in \{1 \ldots m\}$, a_i, b_j, c are integers, each constraint can be constructed in the following form.

$$a_1 \times t_1 + \ldots + a_n \times t_n + b_1 \times p_1 + \ldots + b_m \times p_m \sim c$$

For instance, given the maximum message lifetime p_m and the minimal network latency p_n, when a message generated at t is received at t', $t' - t \leq p_m$ can be a timing constraint used by the receiver to check message freshness and $t' - t \geq p_n$ can be a timing constraint enforced by the physical environment. Additionally, the configuration $L = \mathcal{CS}(p_1, p_2, \ldots, p_m)$ is a set of linear constraints that solely applicable to the timing parameters. In the above example, the constraint (configuration) $p_n > 0$ should be satisfied to model the physical message transmission delay. Before the verification start, L should be specified with an initial configuration, e.g., $p_n > 0$. Whenever a security violation or a function flaw is found during the verification, we update L with new constraints. At last, L is returned as the verification result, which contains the necessary constraints for both system security and functionality. For instance, $p_m \geq p_n$ should be implied by the verification result, because no message could be deliverable in the network otherwise.

As shown in Table 1, 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 nonce generation process '$\nu n.P$' represents that a fresh nonce n is generated and bound to process P. The clock reading process '$\mu t.P$' similarly means that a timestamp t is read from the user's clock and bound to process P. The checking condition c in the conditional process 'if c then P else Q' 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 evaluates to true, process P is executed; otherwise, Q is executed. The timing delay process '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 'let $x = f(m_1, \ldots, m_n)$ 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 variable x in process P; otherwise, process Q is executed. The channel input '$c(x).P$' means that a message, bound to the variable x, is received from the channel c before executing P. The channel output '$\overline{c}(m).P$' describes that the message m is sent to the channel c before executing process P. The uniqueness checking expression 'check m as unique then P' ensures that the value of m has never been evidenced before at this location, comparing with other process replications. The uniqueness checking is particularly useful to prevent replay attacks in practice.

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

TABLE 2
Cryptographic Function Definitions

Additionally, several special events are introduced, aiming at specifying the security claims.

- Right before the initiator finishes its role in starting the protocol, which is usually indicated by sending the last message, he/she 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/she 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 run with the arguments in m at time t.
- When the protocol participant wants to publish a secret message m, he/she emits an $open(m)$ event before doing so.
- The protocol participant can engage a $secrecy(m)$ event to indicate that the message m should not be known to the adversary until $open(m)$ is emitted explicitly.

We use the $init$, $join$, $accept$ events to specify authentication properties, and use the $secrecy$, $open$ events for secrecy properties respectively. As these events are closely related to the security properties, we explain them in Section 3 with the property definitions.

When a message is destructed in any process other than the function application process and the destruction fails,

the behavior is undefined. Hence, we require that all of the messages (e.g., m, m_i) shown in Table 1 do not contain destructors. In this way, messages can only be destructed in the function application process.

Notations and Definitions. Several widely accepted notations and definitions are adopted as follows. $\sigma = \{x_1 \mapsto m_1, \ldots, x_n \mapsto m_n\}$ 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_1 and m_2, when there exists a substitution σ such that $m_1 \cdot \sigma = m_2$, we say that m_1 can be unified to m_2, denoted as $m_1 \rightsquigarrow_\sigma m_2$; when no substitution σ exists such that $m_1 \rightsquigarrow_\sigma m_2$, we say that m_1 cannot be unified to m_2, denoted as $m_1 \not\rightsquigarrow m_2$. Given two messages m_1 and m_2, if there exists a substitution σ such that $m_1 \cdot \sigma = m_2 \cdot \sigma$, we say m_1 and m_2 are unifiable and σ is an unifier of m_1 and m_2. If m_1 and m_2 are unifiable, the most general unifier of m_1 and m_2 is an unifier σ such that for any unifier σ' of m_1 and m_2 there exists a substitution σ'' such that $\sigma' = \sigma \cdot \sigma''$. The most general unifier of m_1 and m_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 $\langle m_1, m_2, \ldots, m_n \rangle$. The concatenation function '$\odot$' between a tuple and an element is defined as $\langle x_1, \ldots, x_n \rangle \odot y = \langle x_1, \ldots, x_n, y \rangle$. The union function '$\uplus$' between a set and an element is defined as $\{x_1, \ldots, x_n\} \uplus y = \{x_1, \ldots, x_n, y\}$. A variable x is bound to a process P when x is constructed by the function application process '$\mathsf{let}\ x = f(m_1, \ldots)\ \mathsf{then}\ P\ \mathsf{else}\ Q$' or the channel input process '$c(x).P$' as shown in Table 1. When a variable x 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. Notice that all of the processes considered in this work are closed. When x is a tuple in the function application process or the channel input process above, we simply write x as $\langle x_1, x_2, \ldots, x_n \rangle$.

Remarks. Private channels can be simulated by secure tunnels using unbreakable encryption in the public channel. For instance, given a private channel named c_p and a message m transmitted in it, we can treat c_p as a symmetric encryption key unknown to the adversary and write $\overline{c_p}(m)$ as $\overline{c}(enc_s(m, c_p))$, when the symmetric encryption enc_s is unbreakable. We thus do not introduce special syntax to specify private channels in *timed applied π-calculus*.

2.2 Running Example: Wide Mouthed Frog

In the following, we use the Wide Mouthed Frog (WMF) [6] protocol as a running example to illustrate our specification as well as our verification method. 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 p_m to represent the maximum message lifetime, ensuring that every message is received within p_m. By default, we consider the minimal network delay as a parameter p_n. Since p_n is a timing parameter related to the network environment, it is not directly used in the protocol specification. Instead, it is a compulsory delay that applies to all of the network transmissions. In addition, we assume that the network latency is always positive, which makes the initial parameter configuration as $L_0 = \{p_n > 0\}$. Notice that a positive network delay is not compulsory in the protocol specification. However, setting the minimal network latency as $p_n \geq 0$ sometimes results in a misleading conclusion: the protocol is correct if and only if p_n equals to 0. Since the network latency p_n cannot be ensured as 0 in practice, the security protocol is thus proved as insecure. Because this final step of manual deduction is undesirable, we remove it by simply requiring a positive network latency in the first place.

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 at t_a
$A \to S \quad : \langle A, enc_s(\langle t_a, B, k \rangle, key(A)) \rangle$
(2) S receives the request from A at t_s
S checks $\quad : t_s - t_a \leq p_m$
$S \to B \quad : enc_s(\langle t_s, A, k \rangle, key(B))$
(3) B receives the message from S at t_b
B checks $\quad : t_b - t_s \leq p_m$
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 ensures the message freshness by checking the timestamp t_a and accepts her request by sending 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 freshness again and accepts the request from A if the message is received timely. All of the transmitted messages are encrypted under the users' long-term keys that are 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) the adversary controls the participation time of all entities in the protocol, (3) S provides its session key exchange service to all of its registered users and (4) the adversary can register as any user at the server, except for A and B. The precise attacker model employed in our work is discussed in Section 3. 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 communication.

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 user (personated by the adversary), which can be shown as the process P_r below.

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

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 can be specified by the adversary. Then, A generates a session key k and

claims k should be unknown to the adversary. Later, A records the clock reading t_a and emits an $init$ event to indicate the protocol initialization with the protocol arguments m_a at t_a. Notice that m_a will be instantiated in Section 3 according to specific authentication properties. Finally, the message $\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle$ 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, we may believe that they are inaccessible to the adversary.

$$\begin{aligned} P_a \triangleq\ & c(r).\nu k.secrecy(k).\mu t_a.init(m_a)@t_a. \\ & \text{if } r = B \text{ then } \overline{c}(\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle).0 \\ & \text{else } open(k).\overline{c}(\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle).0 \end{aligned}$$

As specified by the process P_s, after the server receives a user's request as a tuple $\langle i, x\rangle$, the current time is recorded as t_s and the key $key(i)$ is used to decrypt x. 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 $t_s - t_i \leq p_m$ is passed, the server then believes its participation in a protocol run at time t_s. Similar to the $init$ event, we specify the argument m_s in Section 3. 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.

$$\begin{aligned} P_s \triangleq\ & c(\langle i, x\rangle).\mu t_s.let\ \langle t_i, r, k\rangle = dec_s(x, key(i))\ then \\ & if\ t_s - t_i \leq p_m\ then\ join(m_s)@t_s \\ & .\overline{c}(enc_s(\langle t_s, i, k\rangle, key(r))).0 \end{aligned}$$

Additionally, as shown in the process P_b, when B receives the message from 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 $t_b - t_s \leq p_m$ is passed, B then believes that the request is sent from A within $2 * p_m$ (as the message freshness checking stacks) and claims the acceptance at time t_b.

$$\begin{aligned} P_b \triangleq\ & c(x).\mu t_b.let\ \langle t_s, i, k\rangle = dec_s(x, key(B))\ then \\ & if\ i = A\ then\ if\ t_b - t_s \leq p_m\ then\ accept(m_b)@t_b.0 \end{aligned}$$

Finally, we have a process P_p that broadcasts the public names of *Alice* and *Bob*.

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

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

$$P \triangleq (!P_r)|(!P_a)|(!P_s)|(!P_b)|(!P_p)$$

3 Timed Security Properties

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

3.1 Adversary Model

We assume that an active attacker exists in the network, whose capability is extended from the Dolev-Yao model [12]. The attacker can intercept all communications, compute new messages, generate new nonces and send the messages he obtained. For computation, he can use all the publicly available functions, e.g., encryption, decryption, concatenation. He can also ask the genuine protocol participants to take part in the protocol whenever he needs to. Comparing our attack model with the Dolev-Yao model, attacking weak cryptographic functions and compromising legitimate protocol participants are allowed. A formal definition of the adversary model in timed applied π-calculus is as follows.

Definition 1. **Adversary Process.** The adversary process is defined as an arbitrary *closed* timed applied π-calculus process S which does not emit special events, i.e. $init$, $join$, $accept$ and $secrecy$.

3.2 Timed Authentication

In a protocol, we often 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 message m stands for the arguments used in the current protocol session and the timestamp t represents the timing of the authentication claim. Examples are given later in this section for different types of authentication.

- The protocol initiator emits $init(m)@t$ when he/she initializes the protocol.
- The protocol responder engages $accept(m)@t$ to claim that he/she finishes the protocol.
- The protocol partners emit $join(m)@t$ to indicate his/her participation in the protocol.

When any event is engaged, it means that the protocol participant believes his/her participation of the corresponding role 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 their knowledge of the protocol execution state cannot be increased after this point.

Based on the $init$, $join$ and $accept$ events, the protocol authentication properties then can be formally specified as event correspondences, i.e., the non-injective and injective timed authentication. Additionally, when particular arguments are specified in the events, their correspondence can be further categorized into an agreement property or a synchronization property.

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 2. **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, given the adversary process S, for every occurrence of an $accept$ event in $P|S$, the corresponding $init$ event and $join$ events in Q_n have occurred before in $P|S$, agreeing on the arguments in the events and the constraints in B.

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

Definition 3. **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; (2) given the adversary process S, for every $init$ 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 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 two query sets $\mathbb{Q}_n$ and $\mathbb{Q}_i$ respectively, we have $inj(\mathbb{Q}_n) = \mathbb{Q}_i$ and $non\_inj(\mathbb{Q}_i) = \mathbb{Q}_n$.

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 that certain common knowledge is established among the protocol participants under the timing restrictions.

Example 1. 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 in different processes of WMF as follows.

$$\begin{aligned} m_a &= \langle A, r, k\rangle && in\ P_a \\ m_s &= \langle i, r, k\rangle && in\ P_s \\ m_b &= \langle i, B, k\rangle && in\ P_b \end{aligned}$$

The non-injective timed agreement then can written as

$$\begin{aligned} Q_{na} = accept(\langle i, r, k\rangle)@t_r \\ \leftarrow\!\!\lfloor\ t_s{-}t_i \le \S p_m \wedge t_r - t_s \le \S p_m\ \rfloor\!\!\vdash \\ init(\langle i, r, k\rangle)@t_i, join(\langle i, r, k\rangle)@t_s. \end{aligned} \quad (1)$$

Similarly, we have the injective one as $Q_{ia} = inj(Q_{na})$.

Timed Synchronization Properties. However, the timed agreement properties do not necessarily guarantee the faithful message transmissions 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 [13], 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 ensure that the messages transmitted in the protocol are untampered, so the message received by the receiver is the message sent from the sender for every network transmission.

Example 2. In WMF, we first specify the arguments of the authentication events as follows to reflect the network communications.

$$\begin{aligned} m_a &= \langle r, \langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle\rangle && in\ P_a \\ m_s &= \langle y, enc_s(\langle t_s, i, k\rangle, key(r))\rangle && in\ P_s \\ m_b &= \langle x\rangle && in\ P_b \end{aligned}$$

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

$$\begin{aligned} Q_{ns} = accept(\langle s2b\rangle)@t_r \\ \leftarrow\!\!\lfloor\ t_s{-}t_i \le \S p_m \wedge t_r - t_s \le \S p_m\ \rfloor\!\!\vdash \\ init(\langle r, a2s\rangle)@t_i, join(\langle a2s, s2b\rangle)@t_s \end{aligned}$$

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 $Q_{is} = inj(Q_{ns})$.

3.3 Secrecy

When a message m satisfies secrecy property, it often means that m cannot be known to the adversary. However, in some specific protocols such as commitment protocols, we need a stronger secrecy property because the secret owner may reveal the secret at some protocol stage intentionally. Hence, we define message secrecy as a conditional property [14], i.e. the message should not be known to the adversary before its owner intentionally reveals it. In the *timed applied π-calculus*, before the secret owner reveals a secret m, he/she must explicitly engages an $open(m)$ event. Then, the secrecy property can be defined as follows.

Definition 4. **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, m cannot be sent to the public channel c before $open(m)$ has been engaged in $P|S$.

4 Timed Logic Rules

In this section, we first introduce *timed logic rules* to specify the timed security protocols, which facilitate efficient verification as shown in Section 5. Then, we define the semantics of the *timed applied π-calculus* based on the *timed logic rules*.

Type	Expression	
Message(m)	$f(m_1, m_2, ..., m_n)$	(function)
	$a[], b[], c[], A[], B[], C[]$	(name)
	$[n], [k], [N], [K]$	(nonce)
	$\mathbb{t}, \mathbb{t}_1, \mathbb{t}_i, \mathbb{t}_n$	(timestamp)
	x, y, z, X, Y, Z	(variable)
Parameter(p)	$\S p$	(parameter)
Constraint(B)	$\mathcal{C}(\mathbb{t}_1, \mathbb{t}_2, \ldots, \mathbb{t}_n$ $, \S p_1, \S p_2, \ldots, \S p_m)$	(timing relation)
Configuration(L)	$\mathcal{C}(\S p_1, \S p_2, \ldots, \S p_m)$	(parameter config)
Event (e)	$init(\star[d], m, \mathbb{t})$	(initialization)
	$join(\star m, \star\mathbb{t})$	(participation)
	$accept(\star[d], m, \mathbb{t})$	(acceptance)
	$open(\star m)$	(opening)
	$leak(\star m)$	(leakage)
	$know(\star m, \mathbb{t})$	(knowledge)
	$new(\star[n], l[])$	(generation)
	$unique(\star u, \star l[], m)$	(uniqueness)
Rule(R)	$[\ G\ ]\ e_1, \ldots, e_n \multimap [\ B\ ] \mapsto e$	(rule)

TABLE 3
Syntax of Timed Logic Rules

4.1 Timed Logic Rule

Analyzing the timed security protocols using the *timed applied π-calculus* directly is unfortunately inconvenient, because of its conditional branches, naming bindings, etc. Hence, in this section, we introduce *timed logic rules* to represent the attack capabilities of the adversary that facilitate efficient protocol analysis. However, since we need to describe the message types without concrete processes, we introduce notations to differentiate constants, nonces, timestamps, variables and parameters as shown in Table 3. (1) The syntax of variables and functions are unchanged. (2) Constants are appended with a pair of square brackets from A to $A[]$. (3) Nonces are put inside of a pair of square brackets from n to $[n]$. (4) Timestamps are written with a blackboard bold font from t to $\mathbb{t}$. (5) Parameters are prefixed from p to $\S p$.

Generally, each capability of the adversary is specified as a timed logic rule in the following form.

$$[\ G\ ]\ e_1, e_2, \ldots, e_n \multimap [\ B\ ] \mapsto e,$$

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. The timed and untimed conditions are extracted from the execution trace from the beginning of the process to the current execution point. We discuss their extraction later. When G is empty, we simply omit '$[G]$' in the rule.

The events represent the things that can occur in the protocol. In the timed logic rules, eight types of events are introduced. Similar to the *timed applied π-calculus*, we have $init$, $join$ and $accept$ events that denote the authentication claims made by the legitimate protocol participants. The $init$, $join$ events are premises and the $accept$ events are conclusions. However, their notations have been changed as follows.

$$init(m)@t \rightarrow init([d], m, \mathbb{t})$$
$$join(m)@t \rightarrow join(m, \mathbb{t})$$
$$accept(m)@t \rightarrow accept([d], m, \mathbb{t})$$

The additional nonce $[d]$ represents the session id, which is specifically introduced to check the injective authentication properties.

In order to verify the secrecy property with event reachability checking, we introduce $leak(m)$ as an opposite event of $secrecy(m)$, standing for the revealing of the secret message m. For every $secrecy(m)$ claimed in the process, we fork a parallel sub-process '$c(x)$.if $x = m$ then $leak(m)$.0'. It receives a message x from the network, compares it with m and claims $leak(m)$ if $x = m$. In this way, we reduce a verification problem of message secrecy to a reachability analysis of $leak$ event. Furthermore, when a secret message m is revealed by its owner with intention, an $open(m)$ event should exist in the rule premises. The $open$ event in the *timed logic rule* has the same meaning and syntax in the *timed applied π-calculus*. For every $open(m)$ event engaged in the process, an $open(m)$ event is added into the rule premises, indicating m is revealed willingly. As a result, if a $leak(m)$ event is reachable without having an $open(m)$ event as its premises, the secrecy property of m is violated.

In addition, as shown in Table 3, we have the following three new events. First, $know(m, \mathbb{t})$ means that the adversary possesses the message m at time $\mathbb{t}$. Because the adversary intercepts all communications over the public channel c, for every network input $c(x)$ at time t, we add $know(x, \mathbb{t}')$ satisfying $\mathbb{t}' \leq \mathbb{t}$ to the rule premises, meaning that the adversary need to know x before time t so as to send it to c at t; for every network output $\overline{c}(m)$ at time t, we construct a rule that concludes $know(m, \mathbb{t}')$ and satisfies $\mathbb{t}' - \mathbb{t} \geq \S p_n$, representing m can be intercepted by the adversary after the network delay $\S p_n$. Second, given a nonce generation process $\nu n.P$, we add $new([n], l[])$ to the rule premises, denoting the generation of nonce $[n]$ at the process location $l[]$. Third, $unique(u, l[], m)$ means that the message u should be instantiated uniquely in different process replications and the uniqueness is checked at the location $l[]$. The pair $\langle u, l[]\rangle$ is thus globally unique, acting as an identification of the current session. In the unique event, m characterizes the session of the unique value u, consisting of the network inputs, generated nonces and read timestamps in the chronological order. A message is guaranteed to be unique among different process replications in the following three cases: (1) when 'check u as unique then P' presents in the process, the message u is a unique value (because only one process replication can pass the checking for one u); (2) given '$\nu n.P$' in the process, the nonce $[n]$ is a unique value; (3) when a message m is unique in the current process replication and '$P|Q$' is the next process, the message m remains as unique in P and Q respectively. The location names are generated by a special function $loc()$, which returns a unique name to represent the current process location.

Since we assume that different nonces must have different values, every rule can have at most one new event for every single nonce. When two new events have the same nonce in a rule, we merge them into a single event. Obviously, we need to merge other events similarly in the following scenarios: $know$ events present in a rule for the same message; $unique$ events have the same unique

value and checking location; *init* or *accept* events have the same session id; etc. Thus, we introduce signature to events as shown in Table 3, event signature can be constructed by concatenating its event name with a sequence of messages that prefixed with '$\star$'. For instance, in the event $unique(\star u, \star l[], m)$, the unique value u and the location $l[]$ is prefixed by $\star$, so its signature is '$unique.u.l[]$', where '.' concatenates and separates the strings.

To provide a better understanding of the timed logic rules, we show three examples as follows.

Example 3. Given that the symmetric encryption function enc_s is public, the adversary can use it to encrypt messages. In order to use this function, the adversary first need to know a message m and a key k for encryption. Then, the encryption function can return the encrypted message $enc_s(m, k)$. Hence, the encryption can be represented as the following rule.

$$know(m,\mathbb{t}_1), know(k,\mathbb{t}_2) \\ \dashv[\ \mathbb{t}_1 \leq \mathbb{t} \wedge \mathbb{t}_2 \leq \mathbb{t}\ ]\mapsto know(enc_s(m,k),\mathbb{t})$$

Notice that the timing constraints means that $enc_s(m, k)$ can only be known to the adversary after m and k are known, following the chronological order. □

Example 4. In WMF, the server provides its key registration service to the public as follows.

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

Then, the server's service can be written as follows.

$$[\ u \neq A[] \wedge u \neq B[]\ ]\ know(u,\mathbb{t}_1) \\ \dashv[\ \mathbb{t} - \mathbb{t}_1 \geq \S p_n\ ]\mapsto know(key(u),\mathbb{t})$$

It means that the adversary can register secret keys at the server using any name other than A or B. □

Example 5. Consider *Bob*'s role in the WMF. He receives a message from the server, records his current time and claims acceptance if the message is as expected.

$$P_b \triangleq c(x).\mu t_b.let\ \langle t_s, i, k\rangle = dec_s(x, key(B))\ then \\ if\ i = A\ then\ if\ t_b - t_s \leq p_m\ then\ accept(m_b)@t_b.0$$

Since the adversary can start the protocol whenever it wants to, we assume that t_b is specified by the adversary. In order to make the acceptance claim, the variable x must be in the form of $enc_s(\langle t_s, A, k\rangle, key(B))$, where $t_b - t_s \leq p_m$. Thus, we have *Bob*'s rule as follows.

$$unique([n_b], bob[], \langle enc_s(\langle \mathbb{t}_s, A[], k\rangle, key(B[])), \mathbb{t}_b, [n_b]\rangle) \\ , new([n_b], bob[]), know(\mathbb{t}_b, \mathbb{t}_b) \\ , know(enc_s(\langle \mathbb{t}_s, A[], k\rangle, key(B[])), \mathbb{t}_1) \\ \dashv[\ \mathbb{t}_1 \leq \mathbb{t}_b \wedge \mathbb{t}_b - \mathbb{t}_s \leq \S p_m\ ]\mapsto accept([n_b], m_b, \mathbb{t}_b)$$

The additional nonce $[n_b]$ is introduced as the session id of P_b. Since $[n_b]$ is a random number that is unique globally, its value can identify the current session, including the network input x, the recorded timestamp t_b, and the generated nonce n_b in the process.

We show how the nonce can identify the session as follows. When two nonces in a single rule have the same value $[n]$ and one of them is generated in P_b, we shall have $new([n], bob[])$ and $new([n], x)$ in the rule. Since they have the same signature $new.[n]$, they must be unifiable with $\{x \mapsto bob[]\}$. So, the other nonce is generated in P_b as well. Then, the corresponding $unique$ events have the same signature and thus must be unifiable. As a result, they are generated in the same process replication. □

4.2 Semantic Definitions of Timed Applied π-Calculus

The timed logic rules facilitate efficient protocol verification because they represent the attack capabilities of the adversary in a straight forward manner. Hence, we define the semantics of *timed applied π-calculus* based on the timed logic rules.

Semantics of Functions. Given a function written in *timed applied π-calculus* in the following form.

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

The *timed logic rules* can be accordingly written as follows.

$$know(m_1,\mathbb{t}_1), know(m_2,\mathbb{t}_2), \ldots, know(m_n,\mathbb{t}_n) \\ \dashv[\ \forall i \in \{1 \ldots n\} : \mathbb{t} - \mathbb{t}_i \in D\ ]\mapsto know(m,\mathbb{t})$$

It means that the adversary can obtain the function result after a certain computation time in D, when he/she knows all the function inputs.

Semantics of Processes. Given a process in *timed applied π-calculus*, its execution forms various context information, including generated nonces, timestamps, security claims, validated conditions and network communications. Thus, we need to keep the track of these execution contexts in order to define its semantics. In general, the context of a process P is a tuple $\langle \mathbb{t}_l, U, M, G, H, B, \sigma\rangle$ where

- $\mathbb{t}_l$ is the most recently generated timestamp before the execution of P. We use it to maintain a chronological order of all generated timestamps, i.e., for any newly generated timestamp $\mathbb{t}$, we have $\mathbb{t}_l \leq \mathbb{t}$.
- U is a set of messages that should be uniquely instantiated in different process replications.
- M represents the execution trace of the current process replication, consisting of network inputs, read timestamps and generated nonces in a chronological order. Since the process is deterministic, the process outputs are excluded from M.
- G is a set of untimed guards that leads to P.
- B is a set of timing constraints that leads to P.
- σ is a substitution that is applicable to P.

Given a process P and its contexts $\langle \mathbb{t}_l, U, M, G, H, B, \sigma\rangle$, the timed logic rules extracted from P can be denoted as $\lfloor P \rfloor \mathbb{t}_l U M G H B \sigma$. These timed logic rules represent the capabilities of the adversary, as illustrated in Section 4.1. Since we target at verifying timed security protocols with an unbounded number of sessions, when a protocol P_0 is specified in the *timed applied π-calculus* as shown in Section 2, the specification and verification are actually based on '$\mu t_0.!P_0$', where t_0 is the starting time of the whole process. Then, the semantic rule generation can be fired as $\lfloor P_0 \rfloor \mathbb{t}_0 \emptyset \langle\rangle \emptyset \emptyset \mathbb{U} \emptyset$.

First, we discuss three types of processes that either terminate the current session or fork sub-sessions. They are

the null process '0', the parallel composition process '$P|Q$' and the replication process '$!P$'. Since the current session is completed when the null process 0 is reached, no rule is defined. Given the parallel composition process '$P|Q$' as the next process, the current process can choose both subsessions to execute. For every unique message u in U, u remains a unique message in P and Q respectively. When infinite process replication '$!P$' is the next process, all unique messages in U become duplicated in all the replications of P. Hence, U becomes empty and other contexts remain the same.

$$\lfloor 0 \rfloor_{\mathbb{t}_l} UMGHB\sigma = \emptyset$$
Given $H' = \{unique(u, loc(), M) \mid u \in U\}$, we have
$$\lfloor P|Q \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} UMG(H \cup H')B\sigma \cup \lfloor Q \rfloor_{\mathbb{t}_l} UMG(H \cup H')B\sigma$$
$$\lfloor !P \rfloor_{\mathbb{t}_l} UMGHB\sigma = \lfloor P \rfloor_{\mathbb{t}_l} \emptyset MGHB\sigma$$

Second, when the nonce or timestamp generation process is encountered, we add it into the execution trace M respectively. For the nonce generation process, we indicate its generation by adding a new event to the premises and insert it into U because nonces are random numbers. For the timestamp generation process, we add a timing constraint to describe the chronological order of timestamps as well as a $know$ event to show that the adversary can control the timing of process execution.

$$\lfloor \nu n.P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} (U \uplus [n])(M \odot [n])G(H \uplus new([n], loc()))B\sigma$$
$$\lfloor \mu t.P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}} U(M \odot \mathbb{t})G(H \uplus know(\mathbb{t}, \mathbb{t}))(B \cap \mathbb{t}_l \leq \mathbb{t})\sigma$$

Third, four conditional expressions exist in the *timed applied π-calculus*. The equivalence checking between messages should be included in G, while the timing constraints should be added to B. The timing delay expression first reads the current timing and then checks the timing constraints. The function application process computes the function result and stores it into a variable. Notice that we do not consider the function application delay in the process, because the computation delay specified in the function definition aims at describing the adversary rather than the legitimate protocol participants. Since we can insert additional timing delay into the process whenever necessary, the protocol specification becomes more flexible and accurate.

Given $H' = \{unique(u, loc(), M) \mid u \in U\}$, we have
$$\lfloor \text{if } m_1 = m_2 \text{ then } P \text{ else } Q \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} UMG(H \cup H')B(\sigma \cdot mgu(m_1, m_2))$$
$$\cup \lfloor Q \rfloor_{\mathbb{t}_l} UM(G \wedge m_1 \neq m_2)(H \cup H')B\sigma$$
$$\lfloor \text{if } B_0 \text{ then } P \text{ else } Q \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} UMG(H \cup H')(B \cap B_0)\sigma X$$
$$\cup (\cup_{c \in B_0} \lfloor Q \rfloor_{\mathbb{t}_l} UMG(H \cup H')(B \cap \neg c)\sigma)$$
$$\lfloor \text{wait until } \mu t : B_t \text{ then } P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}} U(M \odot \mathbb{t})G(H \uplus know(\mathbb{t}, \mathbb{t}))(B \cap B_t \cap \mathbb{t}_l \leq \mathbb{t})\sigma$$
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
$$\lfloor \text{let } x = f(m_1, \ldots, m_n) \text{ then } P \text{ else } Q \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} UMG(H \cup H')B(\sigma \cdot \{x \mapsto m'\} \cdot \sigma')$$
$$\cup \lfloor Q \rfloor_{\mathbb{t}_l} UM$$
$$(G \wedge \langle m'_1, \ldots, m'_n \rangle \not\leftrightsquigarrow \langle m_1, \ldots, m_n \rangle)(H \cup H')B\sigma$$

Fourth, network communications can happen in the *timed applied π-calculus*. For every network input, we record the time when it is received and add a premise event as a requirement for the adversary to know that message. On the other hand, we generate a *time logic rule* for every network output, representing that the output will be known to the adversary when it is sent out.

$$\lfloor c(x).P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}} U(M \odot x)G(H \uplus know(x, \mathbb{t}'))(B \cap (\mathbb{t}_l, \mathbb{t}' \leq \mathbb{t}))\sigma$$
Given $H' = \{unique(u, loc(), M) \mid u \in U\}$, we have
$$\lfloor \bar{c}(m).P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} UMG(H \cup H')B\sigma$$
$$\uplus ([\ G\ ]\ H \cup H' \multimap [\ B \cap \mathbb{t} - \mathbb{t}_l \geq \S p_n\ ] \mapsto know(m, \mathbb{t})) \cdot \sigma$$

Fifth, we can check the uniqueness of messages in the process, which could be particularly useful for preventing replay attacks and thus ensure injective timed authentication. In practice, the uniqueness checking is usually implemented by maintaining a database and comparing the new values with the existing ones.

$$\lfloor \text{check } m \text{ as unique then } P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} (U \uplus m)MGHB\sigma$$

Sixth, three types of authentication events can be engaged in the process. The $join$ event have the same arguments as of 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 properties, we add an additional argument $[d]$ to represent the session id. The $init$ and $join$ events are added into the rule premises. The $accept$ events act as the rule conclusions.

Given $M' = M \odot [d]$, $U' = U \uplus [d]$ and $H' =$
$$\{unique(u, loc(), M') \mid u \in U'\} \uplus new([d], loc())$$
$$\lfloor init(m)@t.P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} U'M'G(H \uplus new([d], loc()) \uplus init([d], m, \mathbb{t}))B\sigma$$
$$\lfloor join(m)@t.P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} UMG(H \uplus join(m, \mathbb{t}))B\sigma$$
$$\lfloor accept(m)@t.P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} U'M'G(H \cup H')B\sigma$$
$$\uplus ([\ G\ ]\ H \cup H' \multimap [\ B\ ] \mapsto accept([d], m, \mathbb{t})) \cdot \sigma$$

Seventh, the last two processes in the timed applied π-calculus is for the secrecy claim. The secrecy property is checked as an absence of information leakage during the verification in Section 5, so a new event $leak(m)$ is introduced as a contradiction against $secrecy(m)$. Additionally, if

an $open(m)$ is engaged before $leak(m)$, we deem the leakage of m as intended by its owner.

$$\lfloor open(m).P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} UMG(H \uplus open(m))B\sigma$$

Given $H' = \{unique(u, loc(), M) \mid u \in U\}$, we have

$$\lfloor secrecy(m).P \rfloor_{\mathbb{t}_l} UMGHB\sigma$$
$$= \lfloor P \rfloor_{\mathbb{t}_l} UMG(H \cup H')B\sigma$$
$$\uplus ([\ G\ ]\ H \cup H' \uplus know(m, \mathbb{t}) \dashv\!\lfloor\ B\ \rfloor\!\mapsto leak(m)) \cdot \sigma$$

Example 6. As shown in Example 5 previously, a timed logic rule can be extracted from P_b manually. In this example, we demonstrate how to extract the same semantic rule from P_b automatically. The extraction procedure is shown in Table 4. Initially, we have $\mathbb{t}_l = \mathbb{t}_0$, $U = G = H = \sigma = \emptyset$, $M = \langle\rangle$ and $B = \mathbb{U}$. Since G remains empty, we simply ignore it in the table. Then, after executing every expression in P_b shown in the second column, we update the execution contexts with new information. For the latest timestamp column $\mathbb{t}_l$, we update it with the new value. For the sets U, G, H and σ, we show the new items (if any) that should be added. For the tuple M, we list the new items that should be appended at the end. For the timing constraint B, we write the new constraints that should be applied. Hence, by following the semantic definition of *timed applied π-calculus* based on *timed logic rules*, we can extract the following rule automatically.

$$unique([n_b], bob[], \langle enc_s(\langle \mathbb{t}_s, A[], k\rangle, key(B[])), \mathbb{t}_b, [n_b]\rangle)$$
$$, new([n_b], bob[]), know(\mathbb{t}_b, \mathbb{t}_b)$$
$$, know(enc_s(\langle \mathbb{t}_s, A[], k\rangle, key(B[])), \mathbb{t}'_1)$$
$$\dashv\!\lfloor\ \mathbb{t}_0, \mathbb{t}'_1 \le \mathbb{t}_1 \le \mathbb{t}_b \wedge \mathbb{t}_b - \mathbb{t}_s \le \S p_m\ \rfloor\!\mapsto$$
$$accept([n_b], m_b, \mathbb{t}_b)$$

After we remove the unrelated timestamps $\mathbb{t}_0$ and $\mathbb{t}_1$, and rename the timestamp $\mathbb{t}'_1$ into $\mathbb{t}_1$, the automatically extracted semantic rule becomes identical to the previously manually constructed rule. □

5 Verification Algorithm

After obtaining the initial *timed logic rules* denoted as $\mathbb{R}_{init}$ from the *timed applied π-calculus* process as shown in Section 4, the satisfaction of the security properties then can be verified using the algorithms provided in this section. Generally, after specifying an initial parameter configuration L_0, our verification method iteratively tries to find attacks and then updates the parameter configuration to remove the attacks. In the end, our approach can either compute the parameter configurations that make the protocol satisfy all of the security properties or report that no secure parameter configuration can be found. Given a rule $R = [\ G\ ]\ H\ \dashv\!\lfloor\ B\ \rfloor\!\mapsto e$, a set of rules $\mathbb{R}$ and a parameter configuration L, we use $\alpha(R, L) = [\ G\ ]\ H\ \dashv\!\lfloor\ B \cap L\ \rfloor\!\mapsto e$ and $\alpha(\mathbb{R}, L) = \{\alpha(R) | R \in \mathbb{R}\}$ to represent the rules under the configuration L.

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 attacks. The verification either proves the correctness of the protocol by providing secure configurations of the parameters (represented as succinct constraints), or reports attacks if no secure parameter configuration can be found. Since verifying security protocols is undecidable [15], our algorithm cannot guarantee termination. However, as shown in Section 6, our algorithm can terminate on most of the evaluated security protocols. Additionally, we adopted an on-the-fly approach that checks the security properties (in the second phase) as soon as a rule is generated (in the first phase), so we could terminate early when no secure parameter configuration can be found. Moreover, limiting the number of protocol sessions is allowed in our framework which would guarantee the termination of our algorithm.

5.1 Rule Basis Construction

Before constructing the rule basis, we need to introduce two operators first:

- Given two rules $R = [G]\ H\ \dashv\!\lfloor\ B\ \rfloor\!\mapsto e$ and $R' = [G']\ H'\ \dashv\!\lfloor\ B'\ \rfloor\!\mapsto e'$, if e and $e_0 \in H'$ can be unified with the most general unifier σ such that $G \cdot \sigma \wedge G' \cdot \sigma$ can be valid, their composition is denoted as $R \circ_{e_0} R' = ([\ G \wedge G'\ ]\ H \cup (H' - \{e_0\})\ \dashv\!\lfloor\ (B \cap B')\ \rfloor\!\mapsto e') \cdot \sigma$.
- Additionally, given the above two rules R and R', we define R implies R' denoted as $R \Rightarrow R'$ when $\exists \sigma, e \cdot \sigma = e' \wedge G' \Rightarrow G \cdot \sigma \wedge H \cdot \sigma \subseteq H' \wedge B' \subseteq B \cdot \sigma$.

The rule basis $\beta(\mathbb{R}_{init})$ is constructed based on the initial rules $\mathbb{R}_{init}$. Firstly, we define $\mathbb{R}_v$ to represent the minimal closure of $\mathbb{R}_{init}$ based on the rule composition and the rule implication as follows. (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 $R = [G]\ H\ \dashv\!\lfloor\ B\ \rfloor\!\mapsto e$, 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, $\mathbb{V}$ is a set of trivially satisfiable events, consisting of all the events except for the $know(m, t)$ event where m is not 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 then calculate the rule basis.

$$\beta(\mathbb{R}_{init}) = \{R \mid R = [G]\ H\ \dashv\!\lfloor B\ \rfloor\!\mapsto e \in \mathbb{R}_v \wedge \forall e' \in H : e' \in \mathbb{V}\}$$

Theorem 1. For any rule R in the form of $[G]\ H\ \dashv\!\lfloor\ B\ \rfloor\!\mapsto e$ 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 $\alpha(\beta(\mathbb{R}_{init}), L)$.

Theorem 1 means that the rules in $\alpha(\mathbb{R}_{init}, L)$ have equivalent derivability comparing with those in

$\lfloor P = P_b \rfloor$	$\mathbb{t}_l(\mathbb{t}_0)$	$U(\emptyset)$	$M(\langle\rangle)$	$H(\emptyset)$	$B(\mathbb{U})$	$\sigma(\emptyset)$
$\lfloor c(x) \rfloor$	$\mathbb{t}_1$		x	$know(x, \mathbb{t}'_1)$	$\mathbb{t}_0, \mathbb{t}'_1 \leq \mathbb{t}_1$	
$\lfloor \mu t_b \rfloor$	$\mathbb{t}_b$		$\mathbb{t}_b$	$know(\mathbb{t}_b, \mathbb{t}_b)$	$\mathbb{t}_1 \leq \mathbb{t}_b$	
$\lfloor let\ \langle t_s, i, k\rangle = dec_s(x, key(B))\ then \rfloor$	$\mathbb{t}_b$					$x \mapsto enc_s(\langle \mathbb{t}_s, i, k\rangle, key(B[]))$
$\lfloor if\ i = A\ then \rfloor$	$\mathbb{t}_b$					$i \mapsto A[]$
$\lfloor if\ t_b - t_s \leq p_m\ then \rfloor$	$\mathbb{t}_b$				$\mathbb{t}_b - \mathbb{t}_s \leq \S p_m$	
$\lfloor accept(m_b)@t_b \rfloor \Longrightarrow$	$\mathbb{t}_b$	$[n_b]$	$[n_b]$	$new([n_b], bob[])\ ,\ unique([n_b], bob[], \langle x, \mathbb{t}_b, [n_b]\rangle)$		

TABLE 4
Automatic Generation of Semantic Rule for P_b

$\alpha(\beta(\mathbb{R}_{init}), L)$. However, 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. In order to prove Theorem 1, we prove the following two lemmas first.

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

Proof. Let $R_o = [\ G_o\ ]\ H_o\ \dashv\!\![\ B_o\ ]\!\!\mapsto e_o$, $R'_o = [\ G'_o\ ]\ H'_o\ \dashv\!\![\ B'_o\ ]\!\!\mapsto e'_o$, $R_t = [\ G_t\ ]\ H_t\ \dashv\!\![\ B_t\ ]\!\!\mapsto e_t$, $R'_t = [\ G'_t\ ]\ H'_t\ \dashv\!\![\ B'_t\ ]\!\!\mapsto e'_t$. There should exist a substitution σ such that $e_t \cdot \sigma = e_o$, $H_t \cdot \sigma \subseteq H_o$, $G_o \Rightarrow G_t \cdot \sigma\ B_t \cdot \sigma \supseteq B_o$; $e'_t \cdot \sigma = e'_o$, $H'_t \cdot \sigma \subseteq H'_o$, $G'_o \Rightarrow G'_t \cdot \sigma$, $B'_t \cdot \sigma \supseteq B'_o$. Assume $R_o \circ_e R'_o = ([\ G_o \wedge G'_o\ ]\ H_o \cup (H'_o - e)\ \dashv\!\![\ B_o \cap B'_o\ ]\!\!\mapsto e'_o) \cdot \sigma'$. We discuss the two cases as follows.

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

Second Case. Since $\forall e' \in H'_t$ such that $e' \cdot \sigma \neq e$, we have $H'_t \cdot \sigma \subseteq H'_o - e$. $H'_t \cdot \sigma \cdot \sigma' \subseteq (H_o \cup (H'_o - e)) \cdot \sigma'$, $B'_t \cdot \sigma \cdot \sigma' \supseteq B'_o \cdot \sigma' \supseteq B_o \cdot \sigma' \cap B'_o \cdot \sigma'$, $G'_t \cdot \sigma \cdot \sigma' \Leftarrow G'_o \cdot \sigma' \Leftarrow G_o \cdot \sigma' \cap G'_o \cdot \sigma'$, and $e'_t \cdot \sigma \cdot \sigma' = e'_o \cdot \sigma'$. Therefore, $R'_t \Rightarrow R_o \circ_e R'_o$. $\square$

Definition 5. **Derivation Tree.** Let $\mathbb{R}$ be a set of closed rules and R be a closed rule (a closed rule is a rule with its conclusion initiated by its premises). Let R be a rule in the form of $[\ G\ ]\ e_1, \ldots, e_n\ \dashv\!\![\ B\ ]\!\!\mapsto e$. R can be derived from $\mathbb{R}$ if and only if there exists a finite derivation tree satisfying the following conditions:

1) edges in the tree are labeled by events;
2) nodes are labeled by the rules in $\mathbb{R}$;
3) if a node labeled by R has incoming edges of $e'_1, \ldots, e'_n$ and an outgoing edge of e', satisfying the untimed condition G' and the timed condition B', then $R \Rightarrow [\ G'\ ]\ e'_1, \ldots, e'_n\ \dashv\!\![\ B'\ ]\!\!\mapsto e'$;
4) the outgoing edge of the root is the event e;
5) the incoming edges of the tree leaves are $e_1, \ldots, e_n$.

Additionally, G is the conjunction of all the untimed conditions in the derivation tree, and B is the conjunction of all the timed conditions in the derivation tree. We name this tree as the derivation tree of R based on $\mathbb{R}$.

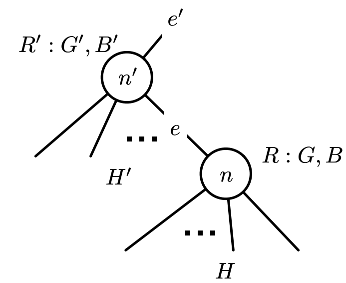

Fig. 1. Two nodes in tree

Lemma 2. For any rule $R = [\ G\ ]\ H\ \dashv\!\![\ B\ ]\!\!\mapsto e$ where $\forall e' \in H : e' \in \mathbb{V}$, R is derivable from $\mathbb{R}_{init}$ if and only if R is derivable from $\beta(\mathbb{R}_{init})$.

Proof. **(only if)** Assuming R is derivable from $\mathbb{R}_{init}$, there exists a derivation tree T_i for S on $\mathbb{R}_{init}$. Since we have $\forall R \in \mathbb{R}_{init}, \exists R' \in \mathbb{R}_v, R' \Rightarrow R$, we can replace all the labels of nodes in T_i with rules in $\mathbb{R}_v$ and get a new derivation tree T_v. Because some of the rules are filtered out from $\mathbb{R}_v$ to $\beta(\mathbb{R}_{init})$ when their premises 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_β is formed so that all the nodes in T_β are labeled by rules in $\beta(\mathbb{R}_{init})$.

To continue our proof, we assume that there exist two nodes n and n' in T_v and they are linked by an edge e_0 as shown in Figure 1. We should have $R, R' \in \mathbb{R}_v$ such that $R \Rightarrow [\ G\ ]\ H\ \dashv\!\![\ B\ ]\!\!\mapsto e$, $R' \Rightarrow [\ G'\ ]\ H'\ \dashv\!\![\ B'\ ]\!\!\mapsto e'$ and $e \in H'$. Because $([\ G\ ]\ H\ \dashv\!\![\ B\ ]\!\!\mapsto e)$ and $([\ G'\ ]\ H'\ \dashv\!\![\ B'\ ]\!\!\mapsto e')$ can be composed on the event e, according to Lemma 1, we could merge these two nodes into one node based on the two different cases given in the proof of Lemma 1. Let $R_o = ([\ G\ ]\ H\ \dashv\!\![\ B\ ]\!\!\mapsto e) \circ_e ([\ G'\ ]\ H'\ \dashv\!\![\ B'\ ]\!\!\mapsto e')$. In the first case, because $\mathbb{R}_v$ is the fixed-point of the service composition, there should exist $R_t \in \mathbb{R}_v$ such that $R_t \Rightarrow R_o$. 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 a rule $R_t \in \mathbb{R}_v$. 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. For every node in T, we prove the rules labeled to the nodes are in $\beta(\mathbb{R}_{init})$ as follows.

- For the leaves of the tree, their incoming edges are labeled by the facts in $\mathbb{V}$. So the leaves are labeled by rules in $\beta(\mathbb{R}_{init})$.

- For an inner node n' of the tree with all its children's rule premises in $\mathbb{V}$. Because n' cannot composed by its children, the premises of the rule labeled to n' should also be in $\mathbb{V}$. So the rules labeled to all the inner nodes are in $\beta(\mathbb{R}_{init})$.

As a consequence, all the nodes in T are labeled by rules in $\beta(\mathbb{R}_{init})$, so R is derivable from $\beta(\mathbb{R}_{init})$.

(if) For every rule in $\mathbb{R}_v$, it should be composed from existing rules, which is in turn composed from $\mathbb{R}_{init}$. Thus all the rules in $\mathbb{R}_v$ should be derivable from $\mathbb{R}_{init}$. In the meanwhile, $\beta(\mathbb{R}_{init})$ does not include any extra rule except for the existing rules in $\mathbb{R}_v$, so $\forall R \in \beta(\mathbb{R}_{init})$, R is derivable from $\mathbb{R}_{init}$. □

Based on the above two lemmas, we can then prove Theorem 1 as follows.

Proof of Theorem 1. 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 Lemma 2, $R = [\ G\ ]\ H \dashv\!\!\!\lfloor\ B\ \rfloor\!\!\!\mapsto e$ 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. □

5.2 Query Searching

In the following, we present how to verify security properties based on $\alpha(\beta(\mathbb{R}_{init}), L)$. A rule disproves non-injective authentication 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.

Definition 6. **Non-injective Authentication Contradiction and Obedience.** A rule $R = [\ G\ ]\ H \dashv\!\!\!\lfloor\ B\ \rfloor\!\!\!\mapsto e$ disproves non-injective authentication $Q_n = accept \leftarrow\!\!\!\lfloor\ B'\ \rfloor\!\!\!\vdash H'$ 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 $\forall \sigma', (H' \cdot \sigma \cdot \sigma' \not\subseteq H \cdot \sigma) \vee (B \cdot \sigma \not\subseteq B' \cdot \sigma \cdot \sigma')$. 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 $\exists \sigma', (H' \cdot \sigma \cdot \sigma' \subseteq H \cdot \sigma) \wedge (B \cdot \sigma \subseteq B' \cdot \sigma \cdot \sigma')$.

Furthermore, an injective authentication is violated if and only if two conditions are satisfied. Firstly, there exists a contradiction to the non-injective version of the query. Secondly, 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. The second condition means that a single *init* event can correspond to two different accept events, which violates the injective authentication property.

Definition 7. **Injective Authentication Contradiction.** Given a pair of rules $\langle R, R' \rangle$, it is a contradiction to the injective authentication query $Q_i = accept \leftarrow\!\!\!\lfloor\ B'\ \rfloor\!\!\!\mapsto init, J'$ 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 without engaging its corresponding *open* event before.

Definition 8. **Secrecy Contradiction.** A rule $R = [\ G\ ]\ H \dashv\!\!\!\lfloor\ B\ \rfloor\!\!\!\mapsto e$ is a contradiction to the secrecy query $Q_s = secrecy(m)$ denoted as $Q_s \nvdash R$ if and only if $G \neq false$, $B \neq \emptyset$, $leak(x) \rightsquigarrow_\sigma e$, $open(x) \cdot \sigma \notin H$ and $\forall e' \in H : e' \in \mathbb{V}$.

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

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

Algorithm 1 illustrates the computing process of the largest L. From line 8 to line 15, we update the parameter configurations to remove the contradictions of the non-injective authentication queries and the secrecy queries (the first condition). From line 16 to line 23, when we find a pair of rules that is a contradiction to an injective authentication query, we find the configurations that can remove the contradiction and update them into the global configurations. (the second condition). From line 24 to line 28, we ensure that every non-injective authentication query has at least one obedience rule (the third condition).

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 2. **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 1, they are derivable from $\alpha(\mathbb{R}_{init}, L)$.

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: $\mathbb{Q}_s$ - the secrecy queries
Output: $\mathbb{L}$ - a set of parameter configurations
$\mathbb{L} = \{L_0\}$;
for $Q \in \mathbb{Q}_n \cup \mathbb{Q}_s \cup non\_inj(\mathbb{Q}_i), L \in \mathbb{L}, R = [\ G\ ]\ H \dashv [\ B\ ] \mapsto e \in \alpha(\beta(\mathbb{R}_{init}), L)$ **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 $Q \in \mathbb{Q}_i, L \in \mathbb{L}, R = [\ G\ ]\ H \dashv [\ B\ ] \mapsto e \in \alpha(\beta(\mathbb{R}_{init}), L), R' = [\ G'\ ]\ H' \dashv [\ B'\ ] \mapsto e' \in \alpha(\beta(\mathbb{R}_{init}), L)$ **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 $L' : Q \nvdash \langle \alpha(R, L'), \alpha(R', L') \rangle = false$ **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 $\mathbb{L}$;

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 1, 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 $R_t = [\ G_t\ ]\ H_t \dashv [\ B_t\ ] \mapsto e_t$ 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 $G_t \neq false$ and $B_t \neq \emptyset$.

- If Q is a secrecy query, R_t has a $leak(m)$ event as conclusion because $Q \nvdash R$. Additionally, because rules cannot be composed on any $open$ event, $open$ should be absent in R_t as well. 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 $Q = accept \leftarrow [\ B_q\ ] \vdash H_q$ is a non-injective authentication query, e_t should be an accept event. So, R_t should satisfy either $Q \vdash R_t$ or $Q \nvdash R_t$. Suppose we have $Q \vdash R_t$. Since $accept$ must be unifiable to e_t, there exists a substitution σ of e_t and $accept$ satisfying $accept \cdot \sigma = e_t$, and $\exists \sigma', (H_q \cdot \sigma \cdot \sigma' \subseteq H_t \cdot \sigma) \wedge (B_t \cdot \sigma \subseteq B_q \cdot \sigma \cdot \sigma')$. Additionally, incoming edges of the tree root cannot be $init$ or $join$ events, so they should persist in R. Hence, $Q \vdash R$, which violates our precondition that $Q \nvdash R$. 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 1, R and R' are also derivable from $\alpha(\beta(\mathbb{R}_{init}), L)$. So there exist 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 have already proved that R_t and R'_t are obedience rules to $non\_inj(Q)$ 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 $Q \nvdash \langle R_t, R'_t \rangle$ □

6 Case Studies

Our verification framework has been implemented as a tool named Security Protocol Analyzer (SPA) (available at [22]), using C++ with 23K LoC . SPA relies on PPL [10] to check the satisfaction of timing constraints, i.e., in order to tell whether a generated rule is feasible or not. To improve the performance, SPA computes the rule basis on-the-fly by updating the parameter configuration as soon as a rule is generated. Hence, the verification process can terminate early if an attack is found.

We have applied SPA to check multiple security protocols as shown in Table 5. All the experiments are conducted

Protocol	Parameterized[a]	Bounded	$\sharp\mathcal{R}^b$	Result	Time
Wide Mouthed Frog [6]	Yes	No	112	Attack [8]	224ms
Wide Mouthed Frog non-injective [7]	Yes	No	80	Attack	43ms
Wide Mouthed Frog injective	Yes	No	80	Secure	51ms
Kerberos V [11]	Yes	No	39205	Attack	2h31m
Kerberos V (c)	Yes	Yes	876356	Secure	34h23m
Auth Range [3], [4]	Yes	No	53	Secure	32ms
CCITT X.509 (1) [16]	No	No	92	Attack [17]	121ms
CCITT X.509 (1c) [17]	No	No	101	Secure	117ms
CCITT X.509 (3) [16]	No	No	433	Attack [6]	2984ms
CCITT X.509 (3) BAN [6]	No	No	285	Secure	1449ms
NS PK [18]	No	No	123	Attack [19]	133ms
NS PK Lowe [19]	No	No	154	Secure	137ms
NS PK Commitment [20]	No	No	189	Secure	187ms
NS PK Time	Yes	No	378	Secure	181ms
SKEME [21]	No	No	302	Secure	1253ms

a. The network latency parameter is considered by default .
b. The number of rules generated in the verification.

TABLE 5
Experiment Results

using a Mac OS X 10.10.4 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 [6], [7], the Kerberos protocols [11], the distance bounding protocol [3], [4] and the CCITT protocols [6], [16], [17]. Additionally, we analyze the untimed protocols like the Needham-Schroeder [18], [19] and SKEME [21]. Most of the protocols can be verified or falsified 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. Particularly, we have successfully found a new timed attack in Kerberos V [11]. Since Kerberos V is the latest version, we simply refer to it as Kerberos. In the following, we illustrate how SPA works with our running example first and then other protocols.

6.1 Wide Mouthed Frog

After checking WMF described in Section 2, an attack is found for its non-injective timed agreement. The two key rules in $\beta(\mathbb{R}_{init})$ are shown below. Notice that we avoid generating a new nonce for the *init* event by reusing the existing session key $[k]$ as the session id of P_a. Since any nonce generated in the current session can act as its session id, this simplification does not weaken nor strengthen the our method. More importantly, it makes the rules much easier to read. Rule (2) represents the execution trace that the server transmits the request from *Alice* to *Bob* for once. It is obedient to the non-injective timed agreement query (1). However, rule (3) represents a possible execution trace that contradicts the query (1). Comparing with the timing constraint '$t_B \leq t_A + 2 \times \S p_m$' in the query, rule (3) has a weaker timing range '$t_B \leq t_A + 4 \times \S p_m$' if $\S p_m > 0$. 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 process P_s, 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 $\S p_m < \S p_n$ or $\S p_m \leq 0$. However, applying any of these constraints to the initial configuration $\S p_n > 0$ leads to the removal of rule (2), which is the only obedience rule in $\alpha(\beta(\mathbb{R}_{init}), L)$. Hence, an attack is found.

$$
\begin{aligned}
&know(t_a,t_a), know(t_s,t_s), know(t_b,t_b), new([k], alice_g[])\\
&, new([n], bob[]), unique([k], alice_c[], \langle B[],[k],t_a\rangle)\\
&, unique([n], bob[], \langle enc_s(\langle t_s, A[],[k]\rangle, key(B[])), t_b, [n]\rangle)\\
&, \mathbf{init}([k], \langle A[], B[],[k]\rangle, t_a), \mathbf{join}(\langle A[], B[],[k]\rangle, t_s)\\
&\dashv\{\, t_b \leq t_s + \S p_m \leq t_a + 2\times \S p_m,\\
&\quad t_a + 2\times \S p_n \leq t_s + \S p_n \leq t_b, \}\mapsto\\
&\qquad\qquad \mathbf{accept}([n], \langle A[], B[],[k]\rangle, t_b) \qquad (2)\\
&know(t_a,t_a), know(t_s,t_s), know(t'_s,t'_s)\\
&, know(t''_s,t''_s), know(t_b,t_b), new([k], alice_g[])\\
&, new([n], bob[]), unique([k], alice_c[], \langle B[],[k],t_a\rangle)\\
&, unique([n], bob[], \langle enc_s(\langle t_s, A[],[k]\rangle, key(B[])), t_b, [n]\rangle)\\
&, \mathbf{init}([k], \langle A[], B[],[k]\rangle, t_a), \mathbf{join}(\langle A[], B[],[k]\rangle, t_s)\\
&, \mathbf{join}(\langle B[], A[],[k]\rangle, t'_s), \mathbf{join}(\langle B[], A[],[k]\rangle, t''_s)\\
&\dashv\{\, t_b \leq t''_s + \S p_m \leq t'_s + 2\times \S p_m\\
&\qquad \leq t_s + 3\times \S p_m \leq t_a + 4\times \S p_m,\\
&\quad t_a + 4\times \S p_n \leq t_s + 3\times \S p_n\\
&\qquad \leq t'_s + 2\times \S p_n \leq t''_s + \S p_n \leq t_b \}\mapsto\\
&\qquad\qquad \mathbf{accept}([n], \langle A[], B[],[k]\rangle, t_b) \qquad (3)
\end{aligned}
$$

Corrected WMF for Non-injective Timed Agreement. The attack of WMF is caused by the symmetric structure of the messages that are sent and received by the server, so the adversary can send the messages from the server back to the server. Hence, this attack can be defended by inserting two different constants m_1 and m_2 into the messages that are sent and received by the server respectively.

$$
\begin{aligned}
&P_a \triangleq c(r).\nu k.secrecy(k).\mu t_a.init(\langle A[], r, k\rangle)@t_a\\
&\quad .\bar{c}(\langle A, enc_s(\langle t_a, r, k, \mathbf{m_1}\rangle, key(A))\rangle).0\\
&P_s \triangleq c(\langle i, x\rangle).\mu t_s.let\ \langle t_i, r, k, \mathbf{m_1}\rangle = dec_s(x, key(i))\ then\\
&\quad if\ t_s - t_i \leq p_m\ then\ join(\langle i, r, k\rangle)@t_s
\end{aligned}
$$

$.\overline{c}(enc_s(\langle t_s, i, k, \mathbf{m_2}\rangle, key(r))).0$

$P_b \triangleq c(x).\mu t_b.let\ \langle t_s, i, k, \mathbf{m_2}\rangle = dec_s(x, key(B))\ then$
$if\ i = A\ then\ if\ t_b - t_s \leq p_m\ then$
$accept(\langle i, B[], k\rangle)@t_b.0$

Then, 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 $0 < \S p_n \leq \S p_m$ with the following obedience rule.

$$know(t_a, t_a), know(t_s, t_s), know(t_b, t_b), new([k], alice_g[]), new([n], bob[]), unique([k], alice_c[], \langle B[], [k], t_a\rangle), unique([n], bob[], \langle enc_s(\langle t_s, A[], [k], m_2[]\rangle, key(B[])), t_b, [n]\rangle), \mathbf{init}([k], \langle A[], B[], [k]\rangle, t_a), \mathbf{join}(\langle A[], B[], [k]\rangle, t_s) \multimap\!\!\vdash t_b \leq t_s + \S p_m \leq t_a + 2 \times \S p_m, t_a + 2 \times \S p_n \leq t_s + \S p_n \leq t_b, \vdash\!\!\multimap \mathbf{accept}([n], \langle A[], B[], [k]\rangle, t_b) \quad (4)$$

However, given two above rules with the same session key $[k]$, we cannot conclude that they have identical $[n]$. This violates the third condition of Algorithm 1, so 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 the first approach that we proposed, *Bob* can maintain a database that stores the previously used session keys to avoid duplicated requests. When a new request is received, *Bob* checks the new session key $[k]$ against the old ones to ensure that it has not been used before.

$P_b \triangleq c(x).\mu t_b.let\ \langle t_s, i, k, m_2\rangle = dec_s(x, key(B))\ then$
$if\ i = A\ then\ if\ t_b - t_s \leq p_m\ then\ check\ k\ as\ unique$
$then\ accept(\langle i, B[], k\rangle)@t_b.0$

Hence, any session key generated by *Alice* can only be accepted by *Bob* for once (i.e., injective). In the corresponding timed logic rule, an additional premise $unique([k], bob[], \langle enc_s(\langle t_s, A[], [k], m_2[]\rangle, key(B[])), t_b, [n]\rangle)$ is inserted into rule (4). When two rules use the same '$[k]$', the $unique$ events then have the same signature '$unique.[k].bob[]$', leading to the unification of session id '$[n]$'. So, the injective timed agreement can be verified in our framework.

In the second approach [19], we add 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_b and send it back to *Alice* under the newly agreed encryption key k. When Alice receives the nonce n_b, she send $inc(n_b)$ back to *Bob*, where $inc(x)$ increases x by 1.

$P_a \triangleq c(r).\nu k.secrecy(k).\mu t_a$
$.\overline{c}(\langle A, enc_s(\langle t_a, r, k, m_1\rangle, key(A))\rangle).c(x)$
$.let\ n_b = dec_s(x, k).init(\langle A[], r, k\rangle)@t_a$
$.\overline{c}(enc_s(inc(n_b), k)).0$
$P_b \triangleq c(x).\mu t_b.let\ \langle t_s, i, k, m_2\rangle = dec_s(x, key(B))$
$then\ if\ i = A\ then\ if\ t_b - t_s \leq p_m\ then$
$\nu n_b.\overline{c}(enc_s(n_b, k)).c(y).let\ y_b = dec_s(y, k)\ then$
$if\ y_b = inc(n_b)\ then\ accept(\langle i, B[], k\rangle)@t_b.0$

Since *Alice* only replies once, *Bob* then can make sure the authentication is injective. During the verification, the $unique$ event of k ensures that at most one n_b will be accepted by *Alice* and the $unique$ event of n_b ensures that *Bob* will establish at most one session for every n_b. Thus, the injective timed agreement can be proved in our framework.

6.2 Commitment Protocols

In commitment protocols, a nonce are often sent out at the end of the protocol session as a proof to the commitment made previously. Since the nonce generated in the legitimated process is unpredictable to the adversary, the adversary cannot get the proof before it is sent out by the process. Consider the following process P, where s is a secret constant that should not be known to the adversary.

$$P \triangleq \nu n.c(x).\overline{c}(n).if\ x = n\ then\ \overline{c}(s).secrecy(s).0$$

Since P receives the message x before it sends out the nonce n, the checking condition $x = n$ can never be satisfied and the secrecy property of s should be preserved. This process P is initially proposed in [23] to illustrate possible false alarms in ProVerif [24], and later used in [20] as an example of verifying commitment protocols. According to [20], [23], ProVerif returns false alarms because it makes over-approximation to nonces. In order to remove the false alarms, Tom et al. [20] proposed to add phases into the protocol execution. Comparing with their approach, our framework can natively prove the secrecy property of s in the process P, because no abstraction is made to the nonces during the verification. Additionally, we use our method to verify Needham-Schroeder protocol with commitment [20] successfully.

More importantly, since the protocol participants can explicitly engage $open$ events to reveal secret messages, we can model commitment protocols in a more straight forward manner. For instance, we can verify the secrecy property in the following process P', which is infeasible by adding phases in ProVerif [20].

$$P' \triangleq \nu n.secrecy(n).open(n).\overline{c}(n).0$$

To be specific, the following two timed logic rules can be extracted from P'.

$$new([n], gen[]), unique([n], cmt[], \langle [n]\rangle), know([n], t) \multimap\!\!\vdash leak([n]) \quad (5)$$

$$new([n], gen[]), unique([n], out[], \langle [n]\rangle), open([n]) \multimap\!\!\vdash know([n], t) \quad (6)$$

After composing rule (6) to rule (5) on the $know$ event, we find that the $leak([n])$ event is only reachable with the $open([n])$ event engaged before. So the secrecy property of the nonce n in P' is preserved.

6.3 Kerberos

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 complete authentication 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 2, the specification for Kerberos itself can be modeled easily. In order to verify Kerberos comprehensively, we model several keys and timestamps (which could be optional) by following its official document RFC 4120 [11] precisely.

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

In the Kerberos model, 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 t_{ep} of the sub-ticket should be no larger than the end-time of the existing ticket. The Kerberos model is available in [22].

Queries. In order to specify the queries, we define three events as follows.

- 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 a 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 a 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 non-injective timed agreements. 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. Notice that the injective timed agreement is unnecessary in Kerberos because it is intended to allow the users to authenticate themselves to the servers for multiple times by using the same unexpired authorization proof.

$$accept(\langle k, c, s\rangle)@t \leftarrow\!\!\lfloor\, t - t' \leq \S l \,\rfloor\!\!- init_{gen}(\langle k, c, s\rangle)@t' \quad (7)$$

$$accept(\langle k, c, s\rangle)@t \leftarrow\!\!\lfloor\, t - t' \leq \S r \,\rfloor\!\!- init_{auth}(\langle k', c, s'\rangle)@t' \quad (8)$$

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 $init_{auth}(\langle k, C, S\rangle, \mathbb{t}') \dashv\!\lfloor\, t - \mathbb{t}' \leq c * \S l \,\rfloor\!\!\mapsto accept(\langle k, C, S\rangle, t)$ in the rule basis, where $c > 1$. This rule is a contradiction to the query (8) 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.

By using SPA, we have successfully found a security flaw in its specification document RFC 4120 [11]. The attack trace is depicted in Figure 2. Suppose the Kerberos is configured

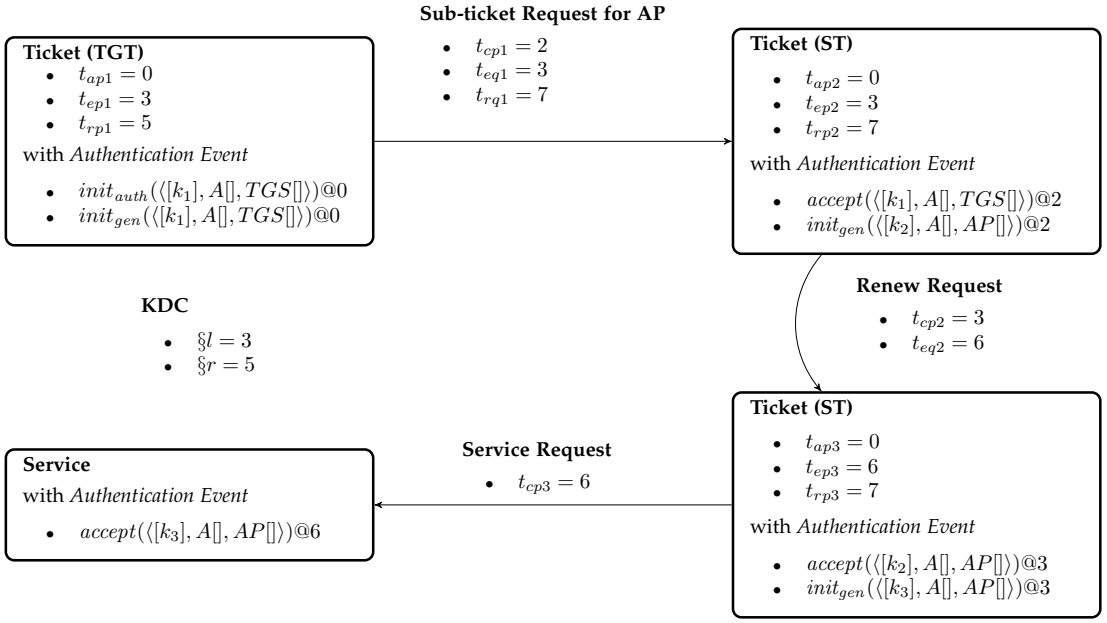

Fig. 2. Attack Found in Kerberos V

with $\S l = 3$ and $\S r = 5$[1], 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 $t_{rq1} - t_{cp1} \leq \S r$. Notice the new sub-ticket's end-time t_{ep2} cannot be larger than the end-time t_{ep1} 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 (8), 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 [25], [26]. An additional check[2] 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 $0 \leq \S l \leq \S r < 2 * \S l$.

1. $\S l$ and $\S r$ are represented by symbols during the verification.

2. 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.

7 Related Works

First of all, this work is a substantial extension to our previous works [1], [2]. In this work, we additionally introduce the *timed applied π-calculus* as an intuitive specification language for timed security protocols. In order to verify the protocols specified in *timed applied π-calculus* automatically, we further define its semantics based on the *timed logic rules* [1], [2]. Furthermore, we extend our framework to verify the injective timed authentication property and stronger secrecy property. During the evaluation, we rewrite all of the existing case studies in [1], [2] using *timed applied π-calculus*. More importantly, we add several new case studies to show our extensions, e.g., the injective version of Wide Mouthed Frog protocol and the commitment protocols. Notice that the commitment protocols, which cannot be handled directly by existing tools like ProVerif [24], can be verified directly by SPA without any modification. The analyzing framework closest to ours was proposed by Delzanno and Ganty [7] which applies $MSR(\mathcal{L})$ to specify unbounded crypto protocols by combining first order multiset rewriting rules and linear constraints. According to [7], 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 [24], [27], [28] are related. However, they are not designed for timed protocols.

Kerberos has been scrutinized over years using formal methods. In [29], 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. [30] using state exploration tool Murφ. They claimed that an attack is found in [31] when two servers exists. However, this attack is actually infeasible in Kerberos's official specification document RFC 1510 [32]. The Kerberos specification RFC 4120 [11] analyzed in this work later superseded RFC 1510. Comparing with the state exploration approach [30], our method can verify protocol with an unbounded number of sessions. Additionally, all above literatures did not consider alternative options supported in Kerberos that may accidentally introduce attacks. Similar to our work, Kerberos V has been analyzed in a theorem proving context by Butler et al. [33]. They took many features into consideration, i.e., the error messages, the encryption types and the cross-realm support. These features are not covered in our work since we focus on the timestamps and timing constraint checking. Meanwhile, our framework can provide intuitive modeling and automatic verifying, whereas Kerberos V is analyzed manually in [33].

8 Conclusions

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 SPA and used it to analyze a wide range of protocols shown in Section 6. 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 SPA to analyze the corrected version of Kerberos, SPA 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.

References

[1] L. Li, J. Sun, Y. Liu, and J. S. Dong, "Tauth: Verifying timed security protocols," in *ICFEM*. Springer, 2014, pp. 300–315.

[2] ——, "Verifying parameterized timed security protocols," in *FM*. Springer, 2015, pp. 342–359.

[3] S. Brands and D. Chaum, "Distance-bounding protocols (extended abstract)," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 765. Springer, 1993, pp. 344–359.

[4] S. Capkun and J.-P. Hubaux, "Secure positioning in wireless networks," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 2, pp. 221–232, 2006.

[5] N. Sastry, U. Shankar, and D. Wagner, "Secure verification of location claims," in *Workshop on Wireless Security*. ACM, 2003, pp. 1–10.

[6] M. Burrows, M. Abadi, and R. M. Needham, "A logic of authentication," *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 18–36, 1990.

[7] G. Delzanno and P. Ganty, "Automatic verification of time sensitive cryptographic protocols," in *TACAS*. Springer, 2004, pp. 342–356.

[8] G. Lowe, "A family of attacks upon authentication protocols," Department of Mathematics and Computer Science, University of Leicester, Tech. Rep., 1997.

[9] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *POPL*, 2001, pp. 104–115.

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

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

[12] D. Dolev and A. C.-C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[13] C. J. F. Cremers, S. Mauw, and E. P. de Vink, "Injective synchronisation: An extension of the authentication hierarchy," *Theor. Comput. Sci.*, vol. 367, no. 1-2, pp. 139–161, 2006.

[14] 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*. Springer, 2014, pp. 11–18.

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

[16] CCITT, "The directory authentication framework - Version 7," 1987, draft Recommendation X.509.

[17] M. Abadi and R. M. Needham, "Prudent engineering practice for cryptographic protocols." *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 6–15, 1996.

[18] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, no. 12, pp. 993–999, 1978.

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

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

[21] H. Krawczyk, "Skeme: a versatile secure key exchange mechanism for internet," in *NDSS*. IEEE CS, 1996, pp. 114–127.

[22] "SPA tool and experiment models," available at http://www.comp.nus.edu.sg/~li-li/r/time.html.

[23] M. Abadi and B. Blanchet, "Analyzing security protocols with secrecy types and logic programs," *J. ACM*, vol. 52, no. 1, pp. 102–146, 2005.

[24] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *CSFW*. IEEE CS, 2001, pp. 82–96.

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

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

[27] C. Cremers, "The Scyther tool: Verification, falsification, and analysis of security protocols," in *CAV*. Springer, 2008, pp. 414–418.

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

[29] G. Bella and L. C. Paulson, "Kerberos version 4: Inductive analysis of the secrecy goals," in *ESORICS*. Springer, 1998, pp. 361–375.

[30] J. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using Murφ;," in *S&P*, 1997, pp. 141–151.

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

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

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