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, critical credentials are often associated with expiry dates in system designs. However, using time correctly in protocol design is challenging, due to the 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 timing parameters whose values cannot be decided in the protocol design stage. In this work, we first propose *timed applied π-calculus* as a formal language for specifying timed security protocols. It supports modeling 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, our method either produces a constraint on the timing parameters which guarantees the security property satisfied by the protocol, or reports an attack that works for any parameter value. 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; or 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 maximum 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 the exact 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 maximum message lifetime p_m. Compared 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. Furthermore, in order to facilitate an efficient verification

- *L. Li and J. Sun are with ISTD, Singapore University of Technology and Design, Singapore 487372. E-mail: {li_li, sunjun}@sutd.edu.sg.*
- *Y. Liu is with the School of Computer Engineering, Nanyang Technological University, Singapore 639798. E-mail: yangliu@ntu.edu.sg.*
- *M. Sun is with the School of Mathematical Sciences, Peking University, Beijing 100080, China. E-mail: sunmeng@math.pku.edu.cn.*
- *J.-S. Dong is with the School of Computing, National University of Singapore, Singapore 487372. E-mail: yangliu@ntu.edu.sg.*

Manuscript received 28 Aug. 2015; revised 6 May 2017; accepted 26 May 2017. Date of publication 5 June 2017; date of current version 21 Aug. 2018. (Corresponding author: Li Li.)
Recommended for acceptance by T. Bulton.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2712621

method, it must have a concise and compact low-level semantics. In this way, the timed security protocol 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 check 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 is also modeled using a set of *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.

Using the *timed logic rules*, we develop our verification algorithms against different security properties in Section 5. The verification result is (1) either an attack that breaks the security property for any parameter value (2) or a constraint that must be satisfied by the parameters to ensure the security of the protocol. 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 SPA 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 present 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 Section 3, we formally defined authentication and secrecy properties based on the events and processes introduced in *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 if 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 and discuss future works in Section 8.

TABLE 1
Syntax of Timed Applied π-Calculus

Type	Expression	
Message(m)	$f(m_1, m_2, \ldots, m_n)$	(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)
	insert m into db 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.*

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

Compared with the *applied π-calculus*, generating, checking and encoding timestamps are allowed in *timed applied π-calculus*. The syntax of *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

TABLE 2
Cryptographic Function Definitions

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)
	$\forall i \in \{1n\}$: $get_n^i(tuple_n(m_1,\ldots,m_n)) \Rightarrow m_i$	(extract tuple)

constants (e.g., the maximum message lifetime p_m) and persistent environment settings (e.g., the minimal network latency 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 instance, the symmetric encryption function enc_s is 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. 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. Then, we can simply write the definition of symmetric encryption 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 act as guard conditions and timing assumptions in the protocol. Generally, each constraint can be constructed as

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

where $\sim \in \{<, \leq\}$ and for any $i \in \{1 \ldots n\}, j \in \{1 \ldots m\}$, a_i, b_j, c are integers. 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 environment. Additionally, the configuration $L = \mathcal{CS}(p_1, p_2, \ldots, p_m)$ is a set of linear constraints constituted by only 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. Afterwards, 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 the 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 channel name c can be any message, e.g., *names*, *function applications* and *nonces*. In this work, we use c_0 as the default public channel name. The unique value insertion expression 'insert m into db as unique then P' is an atomic operation that inserts a message m uniquely into a database named after db. The database can use any message as its name, similar to the channel name. This expression atomically ensures that (1) m does not exist in db before this expression and (2) m is inserted into db after this expression. The unique value insertion expression is particularly useful to prevent replay attacks in practice.

Additionally, the following special events are introduced to specify the security claims.

- Right before the initiator finishes its role in starting the protocol, which is usually indicated by sending the last message, it emits $init(m)@t$ to indicate its belief (according to the protocol) such that a session has been initiated using the arguments m at time t.
- When the responder finishes the protocol successfully, it engages $accept(m)@t$ to indicate its belief such that the protocol is accepted under the arguments in m at time t.
- When other participants join the protocol, they can engage $join(m)@t$ to show their participations in the protocol run with the arguments in m at time t.

- The protocol participant can engage $secrecy(m)$ to indicate that the message m is a secret that should not be known to the adversary unless it is explicitly revealed with by its owner.
- When the protocol participant intends to to publish a message m that has been claimed as secret with $secrecy(m)$, it can explicitly emit $open(m)$ before revealing m.

In this work, we verify security protocols against authentication properties and secrecy properties, which are elaborated in Section 3. The authentication properties are specified with the $init$, $join$ and $accept$ events. The secrecy properties are specified with the $secrecy$ and $open$ events.

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 by the function f 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 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 'let $x = f(m_1, \ldots)$ then P 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 on Processing Time. In this work, every process defined in Table 1 can take arbitrary time to complete. For instance, given $P \triangleq \nu k.c_0(m).\overline{c_0}(enc_s(m,k)).0$ where c_0 is a public channel, the time consumed between the operations is unknown. If k is generated at t_1, m is received at t_2 and $enc_s(m,k)$ is sent at t_3, we only have their order preserved by the constraint $t_1 < t_2 < t_3$. This is because in practice the operation time can be affected by many runtime factors such as network latency, computing power and execution context switch. Similarly, $init$, $join$ and $accept$ events can have delays. As a result, the users need to specify the timing using timestamps in these events to indicate their beliefs of the protocol engagements explicitly.

Remarks on Channels. Following *applied π-calculus* [12], any message, e.g., *names*, *function applications*, *nonces*can be used as channel names in *timed applied π-calculus*. However, we recommend the users to specify their network communications using one public channel (public name) for the following reasons. (1) Private channels used in practice are often built with cryptography (encryptions and keys) using one generic network, e.g., wifi networks and internet. Secret keys with cryptography can be used as *named private channels*, and publicly known names can be used as *named public channels*. For instance, the protocol participants can use $enc_s(\langle m, n \rangle, k)$ to securely transmit the message m, where k is a pre-shared symmetric session key, enc_s is a symmetric encryption function and n is a fresh nonce (as the salt value). In order to build this private channel, the protocol participants could either use existing private channels or key exchange protocols to exchange the key. (2) Explicitly building private channels with cryptography in the protocol models can ensure that the method for building the channels does not introduce security flaws to the protocol in verification. For instance, if the same asymmetric function is used in both of the security protocol and the channel establishment, they may interfere with each other and allow security attacks. (3) In order to provide a strong security guarantee to the verified security protocols, we assume that the network traffics in all of the channels are observable to the adversary.[1]

2.2 Running Example: Wide Mouthed Frog

In the following, we use the Wide Mouthed Frog [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 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 is unlikely to be 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.

1. Based on the adversary model (its capabilities) introduced in Section 3.1, the communications in unobservable channels are completely transparent to the adversary. Hence, if an error is introduce by unobservable channels, it is not considered as an attack from the adversary, which is similar to other security protocol verification works, e.g., [13], [14]. More importantly, considering channels as observable in general can find strictly more security attacks if they exist. Hence, we assume that all of the channels are observable to the adversary in this work. If some unobservable channels are used in the protocol for special purpose, we assume that the protocol designers have ensured their correctness. Then, we can model the messages transmitted in these channels as correctly delivered.

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 is written as $key(u)$, where key is a secret function. WMF then can be informally described as the following three steps.

(1) A generates a random session key k at t_a
$A \rightarrow S$: $\langle A, enc_s(\langle t_a, B, k\rangle, key(A))\rangle$
(2) S receives the request from A at t_s
S checks : $t_s - t_a \leq p_m$
$S \rightarrow B$: $enc_s(\langle t_s, A, k\rangle, key(B))$
(3) B receives the message from S at t_b
B checks : $t_b - t_s \leq p_m$
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_0 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

$$P_r \triangleq c_0(u).if\, u \neq A \wedge u \neq B\ then\ \overline{c_0}(key(u)).0.$$

In WMF, A takes a role of the initiator as specified by P_a below. She first starts the protocol by receiving a responder's name r from c_0, 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

$$\begin{aligned} P_a \triangleq\ & c_0(r).\nu k.secrecy(k).\mu t_a.init(m_a)@t_a. \\ & \text{if } r = B \text{ then } \overline{c_0}(\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle).0 \\ & \text{else } open(k).\overline{c_0}(\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle).0. \end{aligned}$$

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

$$\begin{aligned} P_s \triangleq\ & c_0(\langle i, x\rangle).\mu t_s.let \langle t_i, r, k\rangle = dec_s(x, key(i))\ then \\ & if\ t_s - t_i \leq p_m\ then\ join(m_s)@t_s \\ & .\overline{c_0}(enc_s(\langle t_s, i, k\rangle, key(r))).0. \end{aligned}$$

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

$$\begin{aligned} P_b \triangleq\ & c_0(x).\mu t_b.\ let\ \langle t_s, i, k\rangle = dec_s(x, key(B))\ then \\ & if\ i = A\ then\ if\ t_b - t_s \leq p_m\ then\ accept(m_b)@t_b.0. \end{aligned}$$

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

$$P_p \triangleq \overline{c_0}(A).\overline{c_0}(B).0.$$

The overall process P is the 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 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 [15]. The attacker can intercept all communications, compute new messages, generate new nonces and send the obtained messages. For computation, it 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 K which does not emit special events, i.e., $init$, $join$, $accept$, secrecy and open.*

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 presented 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.

The occurrence of an event 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*

$$Q_n = accept \leftarrow [\ B\ ]{-}\ init, join_1, \ldots, join_n,$$

is satisfied by a closed process P, if and only if, given the adversary process K, for every occurrence of an $accept$ event in $P|K$, the corresponding $init$ event and $join$ events in Q_n have occurred before in $P|K$, 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*

$$Q_i = accept \leftarrow [\ B\ ] \rightarrow init, join_1, \ldots, join_n,$$

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

$$Q_n = accept \leftarrow [\ B\ ]{-}\ init, join_1, \ldots, join_n,$$

is satisfied by P; (2) given the adversary process K, for every $init$ event of Q_i occurred in $P|K$, at most one $accept$ event can occur in $P|K$, agreeing on the arguments in the events and the constraints in B.

For simplicity, given a non-injective query $Q_n = accept \leftarrow [B\ ]{-}\ H$ and its injective version $Q_i = accept \leftarrow [\ B\ ] \rightarrow H$, 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:

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

The non-injective timed agreement then can be written as

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

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 [16], 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:

$$\begin{aligned} Q_{ns} = {} & accept(\langle s2b\rangle)@t_r \\ & \leftarrow\!\![\ t_s - t_i \le p_m \wedge t_r - t_s \le p_m\ ]\!\!\rightarrow \\ & \quad init(\langle r, a2s\rangle)@t_i, join(\langle a2s, s2b\rangle)@t_s. \end{aligned}$$

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 special protocols such as commitment protocols [17], [18], 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 [19], i.e., the message should not be known to the adversary before its owner intentionally reveals it. Hence, in *timed applied π-calculus*, before the secret owner reveals a secret m, he/she must explicitly engages an $open(m)$ event. The secrecy property can be clearly illustrated and motivated by commitment protocols [18] anaylzed in Section 6.2. In this work, 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 K, m cannot be sent to the public before $open(m)$ has been engaged in $P|K$.*

Remarks on Secrecy and Open *Claims.* In this work, comparing with the secrecy property defined in [12], we additionally introduced two *secrecy* and *open* events to specified the secrecy property in the processes. In the following, we illustrate the reasons of introducing these events using the protocol role of *Alice* in WMF as an example. Based on the original protocol specification of WMF, the protocol role of the initiator *Alice*should be specified as P_{oa} as follows:

$$P_{oa} \triangleq c_0(r).\nu k.\mu t_a.\overline{c_0}(\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle).0.$$

In P_{oa}, (1) when the protocol responder claims to be *Bob* ($r = B$), the *secrecy* property of key k should be preserved, which is the verification goal; (2) when the protocol responder is a user controlled by the adversary, k becomes known to the adversary trivially, which does not affect the correctness of the protocol. Hence, we can conclude the *secrecy* property of k in WMF as: k should not be known to the adversary when the responder claims to be *Bob*, a benign responder. It means that the *secrecy* property is a *conditional* property.

However, the secrecy property defined in [12] (a message m satisfies *secrecy* if and only if m cannot be known to the adversary) is *not conditional*. Hence, the security protocols need to be modified in some manner to reflect this condition. For instance, in [13], the specified protocol can only be finished by benign users, and the protocol role of *Alice* in WMF needs be changed to P_{ma} as follows:

$$\begin{aligned} P_{ma} \triangleq {} & c_0(r).\nu k.\mu t_a. \\ & \textbf{if } \mathbf{r = B} \textbf{ then } \overline{c_0}(\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle).0 \textbf{ else } \mathbf{0}. \end{aligned}$$

In P_{ma}, the adversary cannot obtain the messages $\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle$ when $r \neq B$. Since they should be known to the adversary in the original WMF protocol, attacks could be missed.

As a result, we introduce the *secrecy* event in the process calculus to claim the *secrecy* property based on the execution conditions. When a $secrecy(m)$ event is engaged in a process, m instantiated in the current process branch should satisfy the secrecy property. Then, the protocol role of *Alice* in WMF could be modeled as P_{sa}

$$\begin{aligned} P_{sa} \triangleq {} & c_0(r).\nu k.\mu t_a.\text{if } r = B \\ & \text{then } \textbf{secrecy}(\mathbf{k}).\overline{c_0}(\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle).0 \\ & \text{else } \overline{c_0}(\langle A, enc_s(\langle t_a, r, k\rangle, key(A))\rangle).0. \end{aligned}$$

Notice that P_{sa} does not use the *open* event as P_a did in Section 2.2. Both of P_{sa} and P_a are correct specification for the protocol role of *Alice* in WMF when authentication is not considered.

In this work, we introduce the *open* event for additional specification flexibility, which is illustrated by P_a in Section 2.2. The *open* event stands for the explicit revealing behavior from the benign protocol participants. Given a secret message m with $secrecy(m)$ claimed previously, if m is known to the adversary after $open(m)$ has been engaged, the secrecy property of m still holds. This is especially useful where secrets can be revealed to the adversary at a later stage in the protocol. For instance, in some commitment protocols, e.g., [17], a commitment $commit(m, o)$ can be made by a user U to a public message m with a secret open value o. During the protocol execution, $commit(m, o)$ will be sent to the public as a commitment claim, while the open value o should stay as secret at this stage. Later, when U wants to prove that he/she is the one who made the commitment $commit(m, o)$, U can actively reveal the open value o as a proof. Since the open value is not known to the others (including the adversary) before it is revealed in the commitment protocol, it can act as a proof to the commitment. In the commitment protocol, an important property is that the open value o should stay as secret before it is explicitly revealed, which can be specified using the *secrecy* and *open* events with ease. For illustration purpose, a simplified version of the commitment protocol P_{cm} can be modeled as

TABLE 3
Syntax of Timed Logic Rules

Type	Expression	
Message(m)	$f(m_1, m_2, \ldots, m_n)$	(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)	$init(\star[id], m, \mathbb{t})$	(initialization)
	$join(\star[id], m, \mathbb{t})$	(participation)
	$accept(\star[id], m, \mathbb{t})$	(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)	$[\ G\ ]\ e_1, \ldots, e_n - [\ B\ ] \rightarrow e$	(rule)

follows:

$$P_{cm} \triangleq c_0(m).\nu o.\mathbf{secrecy}(\mathbf{o}).\overline{c_0}(commit(m, o)).$$
$$\text{if need\_commitment\_proof then } \mathbf{open}(\mathbf{o}).\overline{c_0}(o).0.$$

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*.

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, name bindings, etc. Hence, in this section, we introduce *timed logic rules* as the semantics of *timed applied π-calculus* 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 with § from p to $\S p$.

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

$$[G]e_1, e_2, \ldots, e_n - [\ B\ ] \rightarrow e,$$

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 are 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, several types of events are introduced as shown in Table 3. Similar to *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:

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

The additional nonce $[id]$ 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 '$c_0(x)$.if $x = m$ then $leak(m).0$', where c_0 is a public channel name. 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 the $leak$ 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 new events. First, $know(m, \mathbb{t})$ means that the adversary possesses the message m at time t. Because the adversary observes all of the channel communications, for every network input '$c(x)$' at time t, we add $know(\langle c, x\rangle, \mathbb{t}')$ satisfying $\mathbb{t}' \leq \mathbb{t}$ to the premises. It means that the adversary needs to know c and x before it can send x to c at time t. Similarly, for every network output '$\overline{c}(m)$' at time t, we construct a rule that concludes $know(m, \mathbb{t}_m)$ with an additional premise of $know(c, \mathbb{t}_c)$, satisfying $\mathbb{t}_m - \mathbb{t} \geq \S p_n \wedge \mathbb{t}_c \leq \mathbb{t}$. It means that the message m transmitted in the channel c at t can be intercepted by the adversary after the network delay $\S p_n$ if it knows the channel name c before t. Second, given a unique value insertion process 'insert m into db as unique then P', we add $unique(m, db, h)$ into the premises means that the message u is a unique value inserted into the database db. The pair $\langle m, db\rangle$ is thus globally unique, acting as an identification of the process replication history h. The process replication history consists of the network inputs, generated nonces and read timestamps in the current process replication in the chronological order. Third, given a nonce generation process '$\nu n.P$', we add both of $new([n], l[])$ and $unique([n], l[], h)$ to the rule premises, denoting the generation of nonce $[n]$ at the process location $l[]$, where h is the process replication history. Notice that the process

replication history h ends when the null process 0 or the replication process $!P$ is reached. The location name $l[]$ is generated by a special function $loc()$, which returns a unique location name for every process. For instance, given

$$P_0 \triangleq !c_0(x).\nu sk.\overline{c_0}(sign(x, sk)).0,$$

where c_0 is a public channel, we can rewrite P_0 as follows:

$$P_0 \triangleq !P_1 \qquad P_1 \triangleq c_0(x).P_2 \qquad P_2 \triangleq \nu sk.P_3$$
$$P_3 \triangleq \overline{c_0}(sign(x, sk)).P_4 \qquad P_4 \triangleq 0.$$

The function $loc()$ returns a unique name for each process P_i where $i \in [0 \ldots 4]$.

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. Similarly, we need to merge other events in the following scenarios: $know$ events presented in a rule for the same message; $unique$ events with the same value and database; $init$, $join$ and $accept$ events with 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 $unique(\star m, \star db, m)$, the message m and the database db is prefixed by $\star$, so its signature is '$unique.m.db$', where '.' concatenates and separates the strings.

To provide a better understanding of the timed logic rules, we show the following examples.

Example 3. For every pubic names in the protocol specification, they should be known to the adversary by default. In WMF, the public channel c_0 is public known. We thus have the following rule

$$-[\ ]\rightarrow know(c_0[], \mathbb{t}).$$

We recommend that the public channel name should be the only public name in the protocol specification. □

Example 4. 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:

$$know(m, \mathbb{t}_1), know(k, \mathbb{t}_2) \\ -[\ \mathbb{t}_1 \leq \mathbb{t} \wedge \mathbb{t}_2 \leq \mathbb{t}\ ]\rightarrowtail know(enc_s(m, k), \mathbb{t}).$$

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

$$P_r \triangleq c_0(u).if\ u \neq A \wedge u \neq B\ then\ \overline{c_0}(key(u)).0.$$

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

$$[u \neq A[] \wedge u \neq B[]]\ know(u, \mathbb{t}_1), know(c_0[], \mathbb{t}_2) \\ -[\ \mathbb{t} - \mathbb{t}_1 \geq \S p_n \wedge \mathbb{t}_2 \leq \mathbb{t}\ ] \rightarrow know(key(u), \mathbb{t}).$$

It means that the adversary can register secret keys at the server over c_0 using any name other than A and B. □

Example 6. 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

$$P_b \triangleq c_0(x).\mu t_b.let\langle t_s, i, k\rangle = dec_s(x, key(B))\ then \\ if i = A\ then\ if\ t_b - t_s \leq p_m\ then\ accept\ (m_b)@t_b.0.$$

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:

$$unique([n_b], bob[], \langle enc_s(\langle \mathbb{t}_s, A[], k\rangle, key(B[])), \mathbb{t}_b, [n_b]\rangle), \\ new([n_b], bob[]), know(\mathbb{t}_b, \mathbb{t}_b), \\ know(\langle c_0[], enc_s(\langle \mathbb{t}_s, A[], k\rangle, key(B[]))\rangle, \mathbb{t}_1) \\ -[\ \mathbb{t}_1 \leq \mathbb{t}_b \wedge \mathbb{t}_b - \mathbb{t}_s \leq \S p_m\ ]\rightarrowtail accept([n_b], m_b, \mathbb{t}_b).$$

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 be used to 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 straightforward 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:

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

It means that the adversary can obtain the function result after a certain 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 $\langle \mathbb{t}_l, f, r, G, H, B, \sigma \rangle$ 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 $\mathbb{t}_l \leq \mathbb{t}$.
- f is a variable representing the full execution trace of the current process that can be identified by the nonces generated in P. The execution trace consists of network inputs, read timestamps and generated nonces in the chronological order. r is a variable representing the rest of the execution trace started from P. For simplicity, we call f and r as the head and the tail of the execution trace until P. In the following, the execution trace of $\langle m_1, m_2 \rangle$ is represented by $\langle m_1, \langle m_2, m_2' \rangle \rangle$. Using this method, we can append message m_3 to the above trace with a substitution $m_2' \mapsto \langle m_3, m_3' \rangle$.
- G is a set of untimed guards that leads to P.
- H is a set of premise events before 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 $\langle \mathbb{t}_l, f, r, G, H, B, \sigma \rangle$, the timed logic rules extracted from P can be denoted as $\lfloor P \rfloor \mathbb{t}_l frGHB\sigma$. 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 $\lfloor P_0 \rfloor \mathbb{t}_0 f_0 f_0 \emptyset \emptyset \mathbb{U} \emptyset$, where f_0 is a variable representing the trace of P_0.

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, nonces generated before $P|Q$ can identify both traces of P and Q; nonces generated in P (Q resp.) can only identify the trace of P (Q resp.). Hence, when r is the trace of $P|Q$, r_p is the trace of P and r_q is the trace of Q, r is mapped to $\langle r_p, r_q \rangle$ in $P|Q$, r is mapped to r_p in P and r is mapped to r_q in Q. When the infinite process replication '$!P$' is the next process, nonces generated before $!P$ cannot identify P; nonces generated in P can identify P. Hence, when r is the trace of $!P$ and r_p is the trace of P, r is mapped to a constant $\perp$ (representing the end of the trace) in $!P$; r and r_p are the same in P

$$
\begin{aligned}
&\lfloor 0 \rfloor \mathbb{t}_l frGHB\sigma = \emptyset \\
&\lfloor P|Q \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l fr_p G(H \cdot \sigma \cdot \{r \mapsto \langle r_p, r_q \rangle\}) B(\sigma \cdot \{r \mapsto r_p\}) \\
&\quad \cup \lfloor Q \rfloor \mathbb{t}_l fr_q G(H \cdot \sigma \cdot \{r \mapsto \langle r_p, r_q \rangle\}) B(\sigma \cdot \{r \mapsto r_q\}) \\
&\lfloor !P \rfloor \mathbb{t}_l frGHB\sigma = \lfloor P \rfloor \mathbb{t}_l frG(H \cdot \sigma \cdot \{r \mapsto \perp\}) B\sigma.
\end{aligned}
$$

Second, when the nonce or timestamp generation process is encountered, we append the nonce or timestamp to the end of the execution trace. For the nonce generation process, we add a new event to H for the nonce generation and insert a $unique$ event to H for its uniqueness, where $loc()$ returns the current location name in the process. 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

$$
\begin{aligned}
&\lfloor \nu n.P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l fr_p G(H \uplus new([n], loc()) \uplus unique([n], loc(), f)) \\
&\quad\quad B(\sigma \cdot \{r \mapsto \langle [n], r_p \rangle\}) \\
&\lfloor \mu t.P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t} fr_p G(H \uplus know(\mathbb{t}, \mathbb{t})) \\
&\quad\quad (B \wedge \mathbb{t}_l < \mathbb{t})(\sigma \cdot \{r \mapsto \langle \mathbb{t}, r_p \rangle\}).
\end{aligned}
$$

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 can be specified more flexibly and accurately in this manner

$$
\begin{aligned}
&\lfloor \text{if } m_1 = m_2 \text{ then } P \text{ else } Q \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l frGHB(\sigma \cdot mgu(m_1, m_2)) \\
&\quad \cup \lfloor Q \rfloor \mathbb{t}_l fr(G \wedge m_1 \neq m_2) HB\sigma \\
&\lfloor \text{if } B_0 \text{ then } P \text{ else } Q \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l frGH(B \wedge B_0)\sigma X \\
&\quad \cup (\cup_{c \in B_0} \lfloor Q \rfloor \mathbb{t}_l frGH(B \wedge \neg c)\sigma) \\
&\lfloor \text{wait until } \mu t : B_t \text{ then } P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t} fr_p G(H \uplus know(\mathbb{t}, \mathbb{t})) \\
&\quad\quad (B \wedge B_t \wedge \mathbb{t}_l < \mathbb{t})(\sigma \cdot \{r \mapsto \langle \mathbb{t}, r_p \rangle\}) \\
&\text{Given function } f \text{ defined as } f(m_1', \ldots, m_n') \Rightarrow m'@D \\
&\text{and } \sigma' = mgu(\langle m_1, \ldots, m_n \rangle, \langle m_1', \ldots, m_n' \rangle), \text{ we have} \\
&\lfloor \text{let } x = f(m_1, \ldots, m_n) \text{ then } P \text{ else } Q \rfloor \mathbb{t}_l UMGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l frHB(\sigma \cdot \{x \mapsto m'\} \cdot \sigma') \\
&\quad \cup \lfloor Q \rfloor \mathbb{t}_l fr(G \wedge \langle m_1', \ldots, m_n' \rangle \not\rightsquigarrow \langle m_1, \ldots, m_n \rangle) HB\sigma.
\end{aligned}
$$

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 $know$ event into the premises, indicating that the adversary must be able to send the message using the channel. Similarly, we generate a *time logic rule* for every network output, representing that the message will be known to the adversary when it is sent using a channel known to the adversary

TABLE 4
Automatic Generation of Semantic Rule for P_b

$\lfloor P = P_b \rfloor$	$\mathbb{t}_l(\mathbb{t}_0)$	$f(f_0)$	$r(f_0)$	$H(\emptyset)$	$B(\mathbb{U})$	$\sigma(\emptyset)$
$\lfloor c(x) \rfloor$	$\mathbb{t}_1$	f_0	r_1	$know(\langle c, x\rangle, \mathbb{t}'_1)$	$\mathbb{t}_0 \le \mathbb{t}_1 \wedge \mathbb{t}'_1 \le \mathbb{t}_1$	$f_0 \mapsto \langle x, r_1\rangle$
$\lfloor \mu t_b \rfloor$	$\mathbb{t}_b$	f_0	r_2	$know(\mathbb{t}_b, \mathbb{t}_b)$	$\mathbb{t}_1 \le \mathbb{t}_b$	$r_1 \mapsto \langle \mathbb{t}_b, r_2\rangle$
$\lfloor let\ \langle t_s, i, k\rangle = dec_s(x, key(B))\ then \rfloor$	$\mathbb{t}_b$	f_0	r_2			$x \mapsto enc_s(\langle \mathbb{t}_s, i, k\rangle, key(B[]))$
$\lfloor if\ i = A\ then \rfloor$	$\mathbb{t}_b$	f_0	r_2			$i \mapsto A[]$
$\lfloor if\ t_b - t_s \le p_m\ then \rfloor$	$\mathbb{t}_b$	f_0	r_2		$\mathbb{t}_b - \mathbb{t}_s \le \S p_m$	
$\lfloor accept(m_b)@t_b \rfloor \Longrightarrow$	$\mathbb{t}_b$	f_0	r_3	$new([n_b], bob[]), unique([n_b], bob[], f_0)$		$r_2 \mapsto \langle [n_b], r_3\rangle$

$$
\begin{aligned}
&\lfloor c(x).P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t} fr_p G(H \uplus know(\langle c, x\rangle, \mathbb{t}')) \\
&\quad\quad (B \wedge \mathbb{t}_l < \mathbb{t} \wedge \mathbb{t}' \le \mathbb{t})(\sigma \cdot \{r \mapsto \langle x, r_p\rangle\}) \\
&\lfloor \overline{c}(m).P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad\quad \uplus ([G]H \uplus know(c, \mathbb{t}_c) \\
&\quad\quad\quad \dashv\!\![\ B \wedge \mathbb{t} - \mathbb{t}_l \ge \S p_n \wedge \mathbb{t}_c \le \mathbb{t}\ ]\!\!\mapsto know(m, \mathbb{t})) \cdot \sigma.
\end{aligned}
$$

Fifth, we can check the uniqueness of messages in the process, which is 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

$$
\begin{aligned}
&\lfloor \text{insert } m \text{ into } db \text{ as unique then } P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l frG(H \uplus unique(m, db, f))B\sigma.
\end{aligned}
$$

Sixth, three types of authentication events can be engaged in the process. In order to check the injective authentication properties, we introduce an additional nonce $[id]$ to represent the session id in the authentication events. The corresponding new and $unique$ events for the new nonce $[id]$ are added as well, where $loc()$ returns the current location name in the process. The $init$ and $join$ events are added into the rule premises. The $accept$ events act as the rule conclusions

$$
\begin{aligned}
&\text{Let } H' = H \uplus new([id], loc()) \uplus unique([id], loc(), f) \\
&\text{and } \sigma' = \sigma \cdot \{r \mapsto \langle [id], r_p\rangle\} \text{ in the following.} \\
&\lfloor init(m)@t.P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l fr_p G(H' \uplus init([id], m, \mathbb{t}))B\sigma' \\
&\lfloor join(m)@t.P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l fr_p G(H' \uplus join([id], m, \mathbb{t}))B\sigma' \\
&\lfloor accept(m)@t.P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l fr_p GH'B\sigma' \\
&\quad\quad \uplus ([G]H' \dashv\!\![\ B\ ]\!\!\mapsto accept([id], m, \mathbb{t})) \cdot \sigma'.
\end{aligned}
$$

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

$$
\begin{aligned}
&\lfloor open(m).P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l frG(H \uplus open(m))B\sigma \\
&\lfloor secrecy(m).P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad = \lfloor P \rfloor \mathbb{t}_l frGHB\sigma \\
&\quad\quad \uplus ([G]H \uplus know(m, \mathbb{t}) \dashv\!\![\ B\ ]\!\!\mapsto leak(m)) \cdot \sigma
\end{aligned}
$$

Example 7. As shown in Example 6 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 $\mathbb{t}_l = \mathbb{t}_0$, $f = r = f_0$, $G = H = \sigma = \emptyset$ 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 $\mathbb{t}_l$, the trace variables f and r, we update them with their new values. For the sets G, H and σ, we show the new elements (if any) that should be added. 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

$$
\begin{aligned}
&unique([n_b], bob[], \\
&\quad \langle enc_s(\langle \mathbb{t}_s, A[], k\rangle, key(B[])), \langle \mathbb{t}_b, \langle [n_b], r_3\rangle\rangle\rangle), \\
&new([n_b], bob[]), know(\mathbb{t}_b, \mathbb{t}_b), \\
&know(\langle c_0[], enc_s(\langle \mathbb{t}_s, A[], k\rangle, key(B[]))\rangle, \mathbb{t}'_1) \\
&\quad\quad \dashv\!\![\ \mathbb{t}_0 \le \mathbb{t}_1 \wedge \mathbb{t}'_1 \le \mathbb{t}_1 \le \mathbb{t}_b \wedge \mathbb{t}_b - \mathbb{t}_s \le \S p_m\ ]\!\!\mapsto \\
&\quad\quad\quad\quad accept([n_b], m_b, \mathbb{t}_b).
\end{aligned}
$$

After we remove the unrelated timestamps $\mathbb{t}_0$ and $\mathbb{t}_1$, rename the timestamp $\mathbb{t}'_1$ into $\mathbb{t}_1$, and rewrite the execution trace from a binary tree $\langle m_1, \langle m_2, \ldots \langle m_n, r\rangle\rangle \ldots\rangle$ into a message tuple $\langle m_1, m_2, \ldots, m_n\rangle$, the automatically extracted semantic rule becomes identical to the previously manually constructed rule. □

Remarks on Execution Traces. In the automatically generated rules, the execution trace $\langle m_1, m_2, \ldots, m_n\rangle$ is represented as a binary tree structure $\langle m_1, \langle m_2, \ldots \langle m_n, r\rangle\rangle \ldots\rangle$ for its extendable nature. For instance, suppose we have a process P_{xs} as follows:

$$
P_{xs} \triangleq c_0(x).\nu s_1.\overline{c_0}(value_1(u, s_1)).\nu s_2.\overline{c_0}(value_2(u, s_2)).0.
$$

When the first value $value_1(u, s_1)$ is sent in P, the execution trace is $trace_1 = \langle x, s_1\rangle$. When the second value $value_2(u, s_2)$ is sent in P, the execution trace is updated to

$trace_2 = \langle x, s_1, s_2 \rangle$. Notice that even though $trace_1$ and $trace_2$ represent the same trace, they are not unifiable because they have different lengths. In order to support the unification between traces with different lengths, we write the trace in the binary tree structure. In this way, $\langle u, \langle s_1, r_1 \rangle \rangle$ and $\langle u, \langle s_1, \langle s_2, r_2 \rangle \rangle \rangle$ become unifiable with $\{r_1 \mapsto \langle s_2, r_2 \rangle\}$.

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 presented 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 in the form of $[G]H -[\ B\ ]\rightarrowtail e$, a set of rules $\mathbb{R}$ and a parameter configuration L, we use $\alpha(R, L) = [G]H -[\ B \wedge L\ ]\rightarrowtail e$ and $\alpha(\mathbb{R}, L) = \{\alpha(R, L) | R \in \mathbb{R}\}$ 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 [20], our algorithm cannot guarantee termination. However, as shown in Section 6, our algorithm terminates on most of the evaluated security protocols, which is similar to other security protocol verification works such as [13], [21], [22]. 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 $R = [G]H -[\ B\ ]\rightarrowtail e$ and $R' = [G']H' -[\ B'\ ]\rightarrowtail e'$, 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 $R \circ_{e_0} R' = ([G \wedge G']H \cup (H' - \{e_0\}) -[\ (B \wedge B')\ ]\rightarrowtail e') \cdot \sigma$.
- 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 $\mathbb{R}_{init}$. First, 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 $R = [G]H - [B\ ] \rightarrow e$, if $\forall e' \in H, e' \in \mathbb{V}$ and $\exists e_0 \notin \mathbb{V}, S \circ_{e_0} S'$ is defined, then $\exists S'' \in \mathbb{R}_v, S'' \Rightarrow R \circ_{e_0} R'$, where $\mathbb{V}$ is a set of events that can be provided by the adversary. In this work, $\mathbb{V}$ is a set of trivially satisfiable events, consisting of the $init$, $join$, $open$, new, $unique$ events, and the $know(v, t)$ event where v is a variable or a timestamp. The third rule means that for any two rules in $\mathbb{R}_v$, if all premises of one rule are trivially satisfiable and their composition exists, their composition is implied by a rule in $\mathbb{R}_v$. Based on $\mathbb{R}_v$, we can then calculate the rule basis

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

First, we define the *derivation tree* as follows. When a derivation tree exists for a rule R based on a set of rules $\mathbb{R}$, we say that R is *derivable* from $\mathbb{R}$.

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 $[G]e_1, \ldots, e_n -[\ B\ ] \rightarrow e$. 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 $R \Rightarrow [G']e'_1, \ldots, e'_n -[\ B'\ ] \rightarrow e'$;
(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 $\mathbb{R}$.

Theorem 1. *For any rule R in the form of $[G]H -[\ B\ ] \rightarrow e$ where $\forall e' \in H : e' \in \mathbb{V}$, R is derivable from $\alpha(\mathbb{R}_{init}, L)$ if and only if R is derivable from $\alpha(\beta(\mathbb{R}_{init}), L)$.*

Theorem 1 means that we can derive the same set of rules from $\alpha(\mathbb{R}_{init}, L)$ and $\alpha(\beta(\mathbb{R}_{init}), L)$. However, since the premises of the rules in $\alpha(\beta(\mathbb{R}_{init}), L)$ are trivially satisfiable by the definition of 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 $R_o = [G_o]H_o -[\ B_o\ ]\rightarrowtail e_o$, $R'_o = [G'_o]H'_o -[\ B'_o\ ]\rightarrowtail e'_o$, $R_t = [G_t]H_t -[\ B_t\ ]\rightarrowtail e_t$, $R'_t = [G'_t]H'_t -[\ B'_t\ ]\rightarrowtail e'_t$. There should exist a substitution σ such that $e_t \cdot \sigma = e_o$,

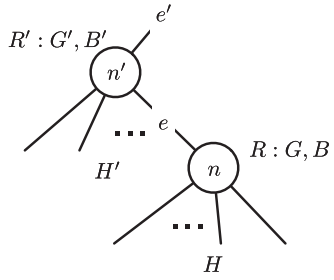


Fig. 1. Two nodes in tree.

$H_t \cdot \sigma \subseteq H_o, G_o \Rightarrow G_t \cdot \sigma B_t \cdot \sigma \supseteq B_o; e'_t \cdot \sigma = e'_o, H'_t \cdot \sigma \subseteq H'_o, G'_o \Rightarrow G'_t \cdot \sigma, \ B'_t \cdot \sigma \supseteq B'_o$. Assume $R_o \circ_e R'_o = ([G_o \wedge G'_o] \ H_o \cup (H'_o - e) \ -[\ B_o \wedge B'_o \ ]\rightarrowtail e'_o) \cdot \sigma'$. 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 $R_t \circ_{e'} R'_t = ([G_t \cdot \sigma_t \wedge G'_t \cdot \sigma_t](H_t \cup (H'_t - e')) \cdot \sigma_t \ -[\ B_t \cdot \sigma_t \wedge B'_t \cdot \sigma_t \ ]\rightarrowtail e'_t \cdot \sigma_t)$. Since $(H_t \cup (H'_t - e')) \cdot \sigma_t \cdot \sigma'_t = (H_t \cup (H'_t - e')) \cdot \sigma \cdot \sigma' \subseteq (H_o \cup (H'_o - e)) \cdot \sigma'$, $e'_t \cdot \sigma_t \cdot \ \sigma'_t = e'_t \cdot \sigma \cdot \sigma' = e'_o \cdot \sigma'$, $(B_t \cdot \sigma_t \wedge B'_t \cdot \sigma_t) \cdot \sigma'_t = B_t \cdot \sigma_t \cdot \sigma'_t \wedge B'_t \cdot \sigma_t \cdot \sigma'_t = B_t \cdot \sigma \cdot \sigma' \wedge B'_t \cdot \sigma \cdot \sigma' \supseteq B_o \cdot \sigma' \wedge B'_o \cdot \sigma')$, and $(G_t \cdot \sigma_t \wedge G'_t \cdot \sigma_t) \cdot \sigma'_t = G_t \cdot \sigma_t \cdot \sigma'_t \wedge G'_t \cdot \sigma_t \cdot \sigma'_t = G_t \cdot \sigma \cdot \sigma' \ \wedge G'_t \cdot \sigma \cdot \sigma' \Leftarrow G_o \cdot \sigma' \wedge G'_o \cdot \sigma'$, 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 $H'_t \cdot \sigma \subseteq H'_o - e$. $H'_t \cdot \sigma \cdot \sigma' \subseteq (H_o \cup (H'_o - e)) \cdot \sigma'$, $B'_t \cdot \sigma \cdot \sigma' \supseteq B'_o \cdot \sigma' \supseteq B_o \cdot \sigma' \wedge B'_o \cdot \sigma'$, $G'_t \cdot \sigma \cdot \sigma' \Leftarrow G'_o \cdot \sigma' \Leftarrow G_o \cdot \sigma' \wedge G'_o \cdot \sigma'$, and $e'_t \cdot \sigma \cdot \sigma' = e'_o \cdot \sigma'$. Therefore, $R'_t \Rightarrow R_o \circ_e R'_o$. □

Lemma 2. *For any rule $R = [G]H \ -[\ B\ ]\rightarrowtail e$ 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 Fig. 1. We should have $R, R' \in \mathbb{R}_v$ such that $R \Rightarrow [G]H \ -[\ B\ ]\rightarrowtail e$, $R' \Rightarrow [G']H' \ -[\ B'\ ]\rightarrowtail e'$ and $e \in H'$. Because $([G]H \ -[\ B\ ]\rightarrowtail e)$ and $([G']H' \ -[\ B'\ ]\rightarrowtail e')$ 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 $R_o = ([G]H \ -[\ B\ ]\rightarrowtail e) \circ_e ([G']H' \ -[\ B'\ ]\rightarrowtail e')$. 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, $R = [G]H \ -[\ B\ ]\rightarrowtail e$ 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 $B \wedge L \neq \emptyset$, $\Gamma(T, L)$ 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 $B \wedge L = \emptyset$, $\alpha(R, L)$ 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 $R = [G]H \ -[\ B\ ]\rightarrowtail e$ disproves non-injective authentication $Q_n = accept \leftarrowtail[\ B'\ ]\vdash H'$ denoted as $Q_n \nvdash R$ if and only if $G \neq false \wedge B \neq \emptyset$, e and accept are unifiable with the most general unifier σ such that $\forall e' \in H, e' \in \mathbb{V}$ and $\forall \sigma', (H' \cdot \sigma \cdot \sigma' \not\subseteq H \cdot \sigma) \vee (B \cdot \sigma \not\subseteq B' \cdot \sigma \cdot \sigma')$. 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. First, there exists a

contradiction to the non-injective version of the query. Second, 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 $Q_i = accept \leftarrow[\ B'\ ]\rightarrowtail init, J'$ 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 $R = [G]H\ -[B\ ]\rightarrowtail e$ is a contradiction to the secrecy query $Q_s = secrecy(m)$ denoted as $Q_s \nvdash R$ if and only if $G \neq false$, $B \neq \emptyset$, $leak(m) \rightsquigarrow_\sigma e$, $open(m)\cdot\sigma \notin H$ 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 three conditions:

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

These three conditions correspond to three "while loops" in Algorithm 1 as follows.

(1) From line 8 to line 15, Algorithm 1 finds the contradiction rule R at line 9 and reduces the size of L to remove R at line 12. After line 15, no contradiction exists for the non-injective authentication queries and the secrecy queries.

(2) From line 16 to line 23, Algorithm 1 finds a rule pair $\langle R, R'\rangle$ that is a contradiction to an injective authentication query at line 17 and reduces the size of L to remove the contradiction at line 20. After line 23, no contradiction exists for the injective authentication queries.

(3) From line 24 to line 28, Algorithm 1 traverses the configurations in $\mathbb{L}$ at line 25 and ensures that at least one obedience rule exists for every non-injective authentication query at line 26. If no obedience rule exists for a non-injective authentication query, the corresponding configuration L is removed. Thus, after line 28, for every L remaining in $\mathbb{L}$, the rules in $\alpha(\beta(\mathbb{R}_{init}), L)$ satisfies all three conditions.

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 $\alpha(\mathbb{R}_{init}, L)$.

Algorithm 1. Parameter Configuration Computation

1: **Input**: $\beta(\mathbb{R}_{init})$ - the rule basis
2: **Input**: L_0 - the initial configuration
3: **Input**: $\mathbb{Q}_n$ - the non-injective authentication queries
4: **Input**: $\mathbb{Q}_i$ - the injective authentication queries
5: **Input**: $\mathbb{Q}_s$ - the secrecy queries
6: **Output**: $\mathbb{L}$ - a set of parameter configurations
7: $\mathbb{L} = \{L_0\}$;
8: **for** $Q \in \mathbb{Q}_n \cup \mathbb{Q}_s \cup non\_inj(\mathbb{Q}_i), L \in \mathbb{L}, R = [G]H\ -[\ B\ ]\rightarrow e \in \alpha(\beta(\mathbb{R}_{init}), L)$ **do**
9: **if** $Q \nvdash R$ **then**
10: $\mathbb{L} = \mathbb{L} - \{L\}$;
11: **for** $L' : B \wedge L' = \emptyset \vee Q \vdash \alpha(R, L')$ **do**
12: $\mathbb{L} = \mathbb{L} \cup \{L \wedge L'\}$;
13: **end for**
14: **end if**
15: **end for**
16: **for** $Q \in \mathbb{Q}_i, L \in \mathbb{L}, R = [G]H\ -[\ B\ ]\rightarrow e \in \alpha(\beta(\mathbb{R}_{init}), L), R' = [G']H'\ -[\ B'\ ]\rightarrow e' \in \alpha(\beta(\mathbb{R}_{init}), L)$ **do**
17: **if** $non\_inj(Q) \vdash R \wedge non\_inj(Q) \vdash R' \wedge Q \nvdash \langle R, R'\rangle$ **then**
18: $\mathbb{L} = \mathbb{L} - \{L\}$;
19: **for** $L' : Q \nvdash \langle \alpha(R, L'), \alpha(R', L')\rangle = false$ **do**
20: $\mathbb{L} = \mathbb{L} \cup \{L \wedge L'\}$;
21: **end for**
22: **end if**
23: **end for**
24: **for** $L \in \mathbb{L}, Q \in \mathbb{Q}_n \cup non\_inj(\mathbb{Q}_i)$ **do**
25: **if** $\nexists R \in \alpha(\beta(\mathbb{R}_{init}), L), Q \vdash R$ **then**
26: $\mathbb{L} = \mathbb{L} - \{L\}$;
27: **end if**
28: **end for**
29: **return** $\mathbb{L}$;

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 $\alpha(\mathbb{R}_{init}, L)$. 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 $R_t = [G_t]H_t\ -[\ B_t\ ]\rightarrowtail e_t$ 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 $Q = accept \leftarrow\!\!\!\!\shortmid [\ B_q\ ]\!\!\!\!\vdash H_q$ 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 $Q \nvdash \langle R_t, R'_t \rangle$ □

Remarks on Non-Termination. Since verifying security protocols is undecidable [20], our algorithm cannot guarantee termination in general. In the following, we describe two common cases that can lead to non-termination.

One common cause of non-termination is the unbounded depth of function application, which is illustrated with Rules (2) and (3) below:

$$know(x, \mathbb{t}_1) \quad -[\ \mathbb{t}_1 \le \mathbb{t}\ ]\!\!\rightarrow \quad know(f(f(x)), \mathbb{t}) \tag{2}$$

$$know(f(x), \mathbb{t}_1) \quad -[\mathbb{t}_1 \le \mathbb{t}\ ]\!\!\rightarrow \quad know(f(f(x)), \mathbb{t}) \tag{3}$$

By composing Rules (2) to (3), we can get

$$know(f(x), \mathbb{t}_1) \ -[\ \mathbb{t}_1 \le \mathbb{t}\ ]\!\!\rightarrow know(f(f(f(x))), \mathbb{t}),$$

which does not imply and is not implied by Rules (2) and (3). By composing Rule (2) to the newly generated rules repeatedly, we can get infinite many rules with increasing application depths of function f. Thus, the verification cannot terminate. Notice that, given a premise event $know(a[], \mathbb{t})$, the events obtainable from Rules (2) and (3) are

$$know(a[], \mathbb{t}),$$
$$know(f(f(a[])), \mathbb{t}),$$
$$know(f(f(f(a[]))), \mathbb{t}),$$
$$\ldots,$$

where $know(f(a[]), \mathbb{t})$ is missing in the list. On the contrary, when $know(f(a[]), \mathbb{t})$ is in the list, Rules (2) and (3) can be replaced with

$$know(x, \mathbb{t}_1) \ -[\ \mathbb{t}_1 \le \mathbb{t}\ ]\!\!\rightarrow know(f(x), \mathbb{t}),$$

which does not have the non-termination problem.

Another common cause of non-termination is the unbounded numbers of freshly generated nonces in a session with unbounded timing constraints, which can be illustrated by Rules (4) and (5) below. Note that $g([n], \mathbb{t})$ has a limited lifetime within $\mathbb{t} + \S d$

$$know(\mathbb{t}_1, \mathbb{t}_1), new([n_1], l_1[]), unique([n_1], l_1[], \langle \mathbb{t}_1, \langle [n_1], r \rangle \rangle) \\ -[\ \mathbb{t}_1 \le \mathbb{t}\ ]\!\!\rightarrow know(g([n_1], \mathbb{t}_1), \mathbb{t}) \tag{4}$$

$$know(g(n_i, \mathbb{t}_i), \mathbb{t}_0), know(\mathbb{t}_{i+1}, \mathbb{t}_{i+1}), new([n_{i+1}], l_2[]), \\ unique([n_{i+1}], l_2[], \langle g(n_i, \mathbb{t}_i), \langle \mathbb{t}_{i+1}, \langle [n_{i+1}], r \rangle \rangle \rangle) \\ -[\ \mathbb{t}_0 \le \mathbb{t} \wedge \mathbb{t}_{i+1} \le \mathbb{t} \\ \wedge\ \mathbb{t}_{i+1} - \mathbb{t}_i \le \S d\ ] \rightarrow know(g([n_{i+1}], \mathbb{t}_{i+1}), \mathbb{t}). \tag{5}$$

Rule (4) means that $g([n_1], \mathbb{t}_1)$ can be known after a nonce $[n_1]$ is generated. Rule (5) means that $g([n_{i+1}], \mathbb{t}_{i+1})$ can be known for another fresh nonce $[n_{i+1}]$ after $g(n_i, \mathbb{t}_1)$ is received within its lifetime $\mathbb{t}_{i+1} - \mathbb{t}_i \le \S d$. We can compose Rules (4) to (5) as the following rule, which does not imply and is not implied by Rules (4) and (5)

$$know(\mathbb{t}_1, \mathbb{t}_1), new([n_1], l_1[]), unique([n_1], l_1[], \langle \mathbb{t}_1, \langle [n_1], r \rangle \rangle), \\ know(\mathbb{t}_2, \mathbb{t}_2), new([n_2], l_2[]), \\ unique([n_2], l_2[], \langle g([n_1], \mathbb{t}_1), \langle \mathbb{t}_2, \langle [n_2], r \rangle \rangle \rangle) \\ -[\mathbb{t}_1 \le \mathbb{t} \wedge \mathbb{t}_2 \le \mathbb{t} \wedge \mathbb{t}_2 - \mathbb{t}_1 \le \S d\ ] \rightarrow know(g([n_2], \mathbb{t}_2), \mathbb{t}).$$

It means that $g([n_2], \mathbb{t}_2)$ can be known after sending the result from Rules (4) to (5). Notice that, comparing with $g([n_1], \mathbb{t}_1)$ in Rule (4), the lifetime of $g([n_2], \mathbb{t}_2)$ is extended. Since the results from the newly composed rules can always be sent to Rule (5), infinite many rules can be generated and thus the verification cannot terminate. In this case, by limiting the application times of Rule (5) in a session, the verification can become terminable. In our future works discussed in Section 8, we plan to develop a practical and generic abstraction method to help the non-terminable cases.

6 Case Studies

Our verification framework has been implemented as a tool named Security Protocol Analyzer (available at [29]), 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 using a Mac OS X 10.10.4 with 2.3 GHz Intel Core i5 and 16G 1333 MHz 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], [23], [24]. Additionally, we analyze the untimed protocols like the Needham-Schroeder [25], [26] and SKEME [28]. Most of the protocols can be verified or falsified quickly for an unbounded

TABLE 5
Experiment Results

Protocol	Parameterized[a]	Bounded	$\sharp\mathcal{R}$[b]	Result	Time
Wide Mouthed Frog [6]	Yes	No	112	Attack [8]	225 ms
Wide Mouthed Frog non-injective [7]	Yes	No	80	Attack	46 ms
Wide Mouthed Frog injective	Yes	No	80	Secure	49 ms
Kerberos V [11]	Yes	No	285	Attack	11.2 s
Kerberos V (c)	Yes	Yes	1,138	Secure	115.9 s
Auth Range [3], [4]	Yes	No	53	Secure	36 ms
CCITT X.509 (1) [23]	No	No	92	Attack [24]	114 ms
CCITT X.509 (1c) [24]	No	No	101	Secure	119 ms
CCITT X.509 (3) [23]	No	No	433	Attack [6]	1,943 ms
CCITT X.509 (3) BAN [6]	No	No	298	Secure	887 ms
NS PK [25]	No	No	123	Attack [26]	94 ms
NS PK Lowe [26]	No	No	150	Secure	89 ms
NS PK Commitment [27]	No	No	179	Secure	114 ms
NS PK Time	Yes	No	173	Secure	174 ms
SKEME [28]	No	No	294	Secure	621 ms

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

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 against the non-injective timed agreement. The two key rules in $\beta(\mathbb{IR}_{init})$ 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 our method. More importantly, it makes the rules much easier to read. Rule (6) 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 (7) represents a possible execution trace that contradicts the query (1). Compared with the timing constraint '$t_b \leq t_a + 2 \times \S p_m$' in the query, rule (7) 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 *Bob* as follows, where K is the adversary.

(1) A generates a random session key k at t_a
$A \rightarrow S$: $\langle A, enc_s(\langle t_a, B, k\rangle, key(A))\rangle$
(2) S receives the request from A at t_s
S checks : $t_s - t_a \leq p_m$
$S \rightarrow B$: $enc_s(\langle t_s, A, k\rangle, key(B))$
(3) K forwards the message to S before $t_s + p_m$
$K \rightarrow S$: $\langle B, enc_s(\langle t_s, A, k\rangle, key(B))\rangle$
(4) S receives the request from K at t'_s
S checks : $t'_s - t_s \leq p_m$
$S \rightarrow A$: $enc_s(\langle t'_s, B, k\rangle, key(A))$
(5) K forwards the message to S before $t'_s + p_m$
$K \rightarrow S$: $\langle A, enc_s(\langle t'_s, B, k\rangle, key(A))\rangle$
(6) S receives the request from K at t''_s
S checks : $t''_s - t'_s \leq p_m$
$S \rightarrow B$: $enc_s(\langle t''_s, A, k\rangle, key(B))$
(7) B receives the message from S at t_b
B checks : $t_b - t''_s \leq p_m$
B accepts the session key k

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 rule, 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 (6), which is the only obedience rule in $\alpha(\beta(\mathbb{IR}_{init}), L)$. Hence, an attack is found

$$\begin{aligned}&\{know\ \&\ new\ \&\ unique\ events\ldots\},\\ &\mathbf{init}([k], \langle A[], B[], [k]\rangle, \mathbb{t}_a), \mathbf{join}(\langle A[], B[], [k]\rangle, \mathbb{t}_s)\\ &\quad \dashv[\ \mathbb{t}_b \leq \mathbb{t}_s + \S p_m \leq \mathbb{t}_a + 2 \times \S p_m,\\ &\qquad \mathbb{t}_a + 2 \times \S p_n \leq \mathbb{t}_s + \S p_n \leq \mathbb{t}_b,]\!\rightarrowtail\\ &\qquad\qquad \mathbf{accept}([n], \langle A[], B[], [k]\rangle, \mathbb{t}_b)\end{aligned} \tag{6}$$

$$\begin{aligned}&\{know\ \&\ new\ \&\ unique\ events\ldots\},\\ &\mathbf{init}([k], \langle A[], B[], [k]\rangle, \mathbb{t}_a), \mathbf{join}([n_s], \langle A[], B[], [k]\rangle, \mathbb{t}_s),\\ &\mathbf{join}([n'_s], \langle B[], A[], [k]\rangle, \mathbb{t}'_s), \mathbf{join}([n''_s], \langle A[], B[], [k]\rangle, \mathbb{t}''_s)\\ &\ \dashv[\ \mathbb{t}_b \leq \mathbb{t}''_s + \S p_m \leq \mathbb{t}'_s + 2 \times \S p_m\\ &\quad \leq \mathbb{t}_s + 3 \times \S p_m \leq \mathbb{t}_a + 4 \times \S p_m,\\ &\mathbb{t}_a + 4 \times \S p_n \leq \mathbb{t}_s + 3 \times \S p_n\\ &\quad \leq \mathbb{t}'_s + 2 \times \S p_n \leq \mathbb{t}''_s + \S p_n \leq \mathbb{t}_b\ ]\!\rightarrowtail\\ &\qquad\qquad \mathbf{accept}([n], \langle A[], B[], [k]\rangle, \mathbb{t}_b).\end{aligned} \tag{7}$$

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 $\mathbf{m_1}$ and $\mathbf{m_2}$ into the messages that are sent and received by the server respectively

$$
\begin{aligned}
&P_a \triangleq c_0(r).\nu k.secrecy(k).\mu t_a.init(\langle A[], r, k\rangle)@t_a\\
&\quad .\overline{c_0}(\langle A, enc_s(\langle t_a, r, k, \mathbf{m_1}\rangle, key(A))\rangle).0\\
&P_s \triangleq c_0(\langle i, x\rangle).\mu t_s.let\langle t_i, r, k, \mathbf{m_1}\rangle = dec_s(x, key(i))\ then\\
&\quad if\ t_s - t_i \le p_m\ then\ join(\langle i, r, k\rangle)@t_s\\
&\quad .\overline{c_0}(enc_s(\langle t_s, i, k, \mathbf{m_2}\rangle, key(r))).0\\
&P_b \triangleq c_0(x).\mu t_b.let\langle t_s, i, k, \mathbf{m_2}\rangle = dec_s(x, key(B))\ then\\
&\quad if\ i = A\ then\ if\ t_b - t_s \le p_m\ then\\
&\quad accept(\langle i, B[], k\rangle)@t_b.0.
\end{aligned}
$$

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 $0 < \S p_n \le \S p_m$ with rule (6). However, given two instances of rule (6) 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 duplicate requests. When a new request is received, *Bob* checks the new session key $[k]$ in database db as unique to ensure that it has not been used before

$$
\begin{aligned}
&P_b \triangleq c(x).\mu t_b.let\langle t_s, i, k, m_2\rangle = dec_s(x, key(B))\ then\\
&\quad if\ i = A\ then\ if\ t_b - t_s \le p_m\ then\\
&\quad insert\ k\ into\ db\ as\ unique\ then\\
&\quad accept(\langle i, B[], k\rangle)@t_b.0.
\end{aligned}
$$

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 $unique([k], db[], \langle enc_s(\langle t_s,\ A[],\ [k],\ m_2[]\rangle,\ key(B[])),\ \langle t_b,\ \langle[n],\ r_b\rangle\rangle\rangle)$ is added. When two rules use the same '$[k]$', the $unique$ events then have the same signature '$unique.[k].db[]$', leading to the unification of session id '$[n]$'. So, the injective timed agreement can be verified in our framework.

In the second approach [26], 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

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

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 and c_0 is a public channel name

$$P \triangleq \nu n.c_0(x).\overline{c_0}(n).if x = n\ then\ \overline{c_0}(s).secrecy(s).0.$$

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 [18] to illustrate possible false alarms in ProVerif [13], and later used in [27] as an example of verifying commitment protocols. According to [18], [27], ProVerif returns false alarms because it makes over-approximation to nonces. In order to remove the false alarms, Tom et al. [27] 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 [27] 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 [27]

$$P' \triangleq \nu n.secrecy(n).open(n).\overline{c_0}(n).0.$$

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

$$
\begin{aligned}
&new([n], gen[]), unique([n], gen[], \langle[n], r\rangle),\\
&\quad know([n], t)\ \dashv[\ ]\vdash leak([n])
\end{aligned}
\tag{8}
$$

$$
\begin{aligned}
&new([n], gen[]), unique([n], gen[], \langle[n], r\rangle),\\
&\quad open([n])\ \dashv[\ ]\vdash know([n], t).
\end{aligned}
\tag{9}
$$

After composing rules (9) to (8) 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 $\mathtt{ap}$ 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 [29].

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

- When an initial ticket is generated at KAS, an $\mathbf{init_{auth}}(\langle k, u, s\rangle)@t$ event is engaged, where k is the fresh session key, u 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 $\mathbf{init_{gen}}(\langle k, u, s\rangle)@t$ event is engaged. Its arguments have the same meaning as those in $init_{auth}$.
- Whenever a ticket is accepted by the server, an $\mathbf{accept}(\langle k, u, s\rangle)@t$ event is engaged, where k is the agreed session key, u 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

$$accept(\langle k, u, s\rangle)@t \leftarrow\!\lfloor\ t - t' \leq \S l\ \rfloor\!- init_{gen}(\langle k, u, s\rangle)@t' \quad (10)$$

$$accept(\langle k, u, s\rangle)@t \leftarrow\!\lfloor\ t - t' \leq \S r\ \rfloor\!- init_{auth}(\langle k', u, s'\rangle)@t'. \quad (11)$$

Verification Results. For the termination of the verification, we need to initially configure the parameters as $\S r < x * \S l$, where x 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 $init_{auth}(\langle k, C, S\rangle, \mathtt{t}') - [\ \mathtt{t} - \mathtt{t}' \leq y * \S l\ ] \rightarrow accept(\langle k, C, S\rangle, \mathtt{t})$ in the rule basis, where $y > 1$. This rule is a contradiction to the query (11) because $\S r$ is not necessarily larger than $y * \S l$. However, Algorithm 1 can add a new constraint $y * \S l \leq \S r$ to the existing configuration and then continue searching. Since we have infinitely many such rules in $\beta(\mathbb{R}_{init})$ with different values of y, 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 Fig. 2. Suppose the Kerberos is configured with $\S l = 3$ and $\S r = 5$,[2] and a user Alice has already

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

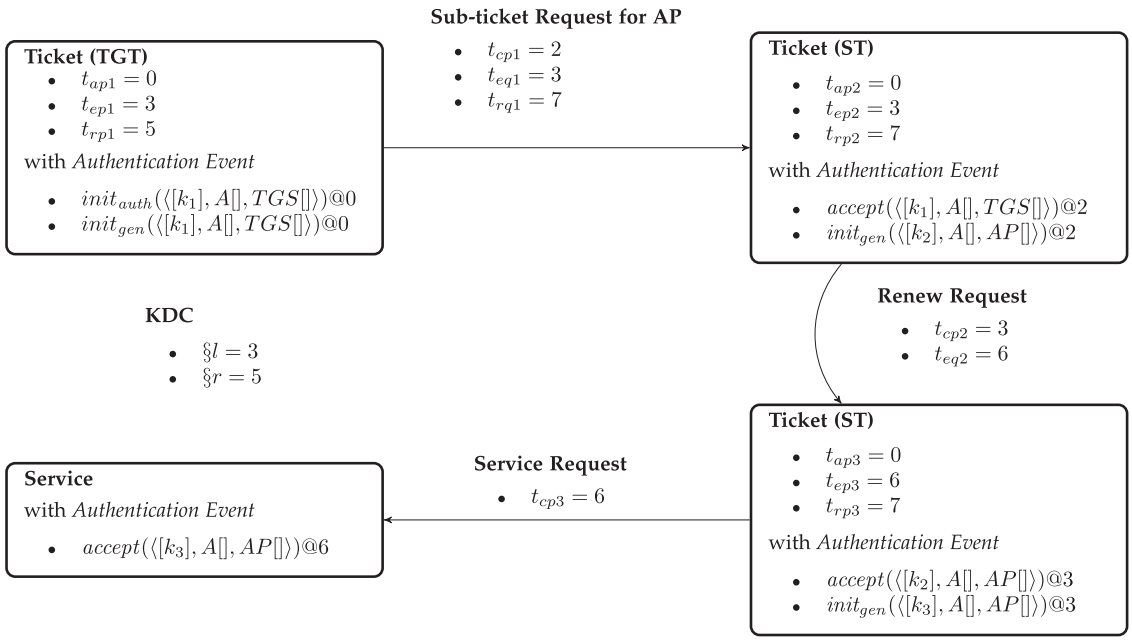


Fig. 2. Attack found in Kerberos V.

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_{cp2} 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 (11), 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 [30], [31]. An additional check[3] has been inserted to regulate that the renewable lifetime in the sub-ticket should be smaller than the renewable lifetime in the existing ticket. We later confirmed with Kerberos team that this is an error in its specification document, which could have led to a security issue but has not done so in its current implementation.

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

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

7 Related Works

We discuss the related works from the following aspects.

Extensions from Previous Works. This work is a substantial extension of [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 and stronger secrecy properties. 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.

Timed Related Security Protocol Verification. 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. Additionally, a verification method was proposed in [32] to verify timed equivalence property for untimed protocols. It verification result ensures that the adversary cannot observe differences between protocol executions considering time. Notice that timed operations, e.g., reading, using and comparing timestamps, cannot be specified using their verification method. Hence, their work is proposed for a different security analysis goal comparing with SPA. Furthermore, [33] was proposed to find timed attacks in cyber-physical security protocols, considering dense time and

processing circle. Comparing with their work, we do not consider the attacks that could be introduced by specific physical properties, like processing circle, at present. Instead, we aims at finding the logic flaws introduced by the cryptography application with time.

Kerberos Verification. Kerberos has been scrutinized over years using formal methods. In [34], 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. [35] using state exploration tool Murφ. They claimed that an attack is found in [36] when two servers exist. However, this attack is actually infeasible in Kerberos's official specification document RFC 1510 [37]. The Kerberos specification RFC 4120 [11] analyzed in this work later superseded RFC 1510. Compared with the state exploration approach [35], our method can verify protocols with an unbounded number of sessions. Additionally, the above literatures do 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. [38]. 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 [38].

Untimed Security Protocol Verifiers. Many tools are proposed to verify untimed protocols, e.g., ProVerif [13], Athena [14], Scyther [22] and Tamarin [21]. ProVerif [13] and our tool SPA both use Horn logic rules to represent the protocol execution. Horn logic reasoning makes the verification process extremely efficient compared with others. ProVerif relies on the nonce abstraction [13]. SPA does not have the nonce abstraction and guarantees partial correctness for its verification result, which makes it suitable for verifying timed protocols and commitment protocols. Athena [14] and Scyther [22] use strand space for the protocol verification, and Tamarin [21] uses multiset rewriting rules. Comparing with our method, strand space and multiset rewriting rules consider the protocol execution as non-monotonic. Hence, they are suitable for specifying compromised adversaries [39] and verifying stateful protocols [40], [41].

Timed Modeling Languages. Many languages have been proposed to model timed systems and protocols, e.g., Timed Automata [42], [43], Timed CSP [44]. Furthermore, many verification tools have been proven successful, e.g., Uppaal [45], KRONOS [46]. However, they are not suitable for modeling timed security protocols for the following reasons. First, they are initially proposed for timed systems and protocols without the adversary. Adding the adversary is non-trivial, e.g., the events and the channel communications do not need to be synchronized in the protocol model considering the presence of the adversary. Furthermore, these models are often verified by model checking for a bounded number of processes using state-space traversal searching algorithm, which is not applicable to security protocol verification that typically requires checking an unbounded number of processes. More importantly, *applied π-calculus* [12] is a widely-used security protocol modeling language in the security community [13], [41], [47]. Hence, we extend *applied π-calculus* with time to model timed security protocols.

8 Conclusions and Future Works

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. Furthermore, comparing with other process algebraic languages, e.g., CSP, other time operations like timeout, interruption are supported. They could be useful in security protocol design and specification. Hence, we plan to extend the *timed applied π-calculus* with these timed operations in our future works. Moreover, considering other related properties, e.g., observational equivalence [32], [48], forward secrecy [49], [50], in the timing domain, could be interesting as well. Hence, we plan to extend SPA with more verification supports for complex security properties and strong adversary models in our future works.

Acknowledgments

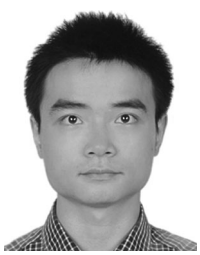
This work is a substantial extension to [1], [2] with the following extra contents. First, we develop an intuitive protocol specification language for timed security protocols. Second, the semantics of our specification language is formally defined using timed logic rules proposed in [1], [2]. Third,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

References

[1] L. Li, J. Sun, Y. Liu, and J. S. Dong, "TAuth: Verifying timed security protocols," in *Proc. Int. Conf. Formal Eng. Methods*, 2014, pp. 300–315.

[2] L. Li, J. Sun, Y. Liu, and J. S. Dong, "Verifying parameterized timed security protocols," in *Proc. Int. Symp. Formal Methods*, 2015, pp. 342–359.

[3] S. Brands and D. Chaum, "Distance-bounding protocols (extended abstract)," in *Proc. Workshop Theory Appl. Cryptographic Techn.*, 1993, pp. 344–359.
[4] S. Capkun and J.-P. Hubaux, "Secure positioning in wireless networks," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 2, pp. 221–232, Feb. 2006.
[5] N. Sastry, U. Shankar, and D. Wagner, "Secure verification of location claims," in *Proc. 2nd Workshop Wireless Secur.*, 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 *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2004, pp. 342–356.
[8] G. Lowe, "A family of attacks upon authentication protocols," Dept. Math. Comput. Sci., Univ. Leicester, Leicester, U.K., 1997.
[9] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Proc. 28th ACM SIGPLAN-SIGACT Symp. Principles Programm. Languages*, 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 *Proc. Int. Static Anal. Symp.*, 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] M. Abadi and B. Blanchet, "Analyzing security protocols with secrecy types and logic programs," *J. ACM*, vol. 52, no. 1, pp. 102–146, 2005.
[13] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *Proc. 14th IEEE Workshop Comput. Secur. Found.*, 2001, pp. 82–96.
[14] D. X. Song, S. Berezin, and A. Perrig, "Athena: A novel approach to efficient automatic security protocol analysis," *J. Comput. Secur.*, vol. 9, no. 1/2, pp. 47–74, 2001.
[15] D. Dolev and A. C.-C. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. IT-29, no. 2, pp. 198–207, Mar. 1983.
[16] C. J. F. Cremers, S. Mauw, and E. P. de Vink, "Injective synchronisation: An extension of the authentication hierarchy," *Theoretical Comput. Sci.*, vol. 367, no. 1/2, pp. 139–161, 2006.
[17] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Proc. 11th Annu. Int. Cryptology Conf.*, 1991, pp. 129–140.
[18] X. Allamigeon and B. Blanchet, "Reconstruction of attacks against cryptographic protocols," in *Proc. IEEE Workshop Comput. Secur. Found.*, 2005, pp. 140–154.
[19] L. Li, J. Pang, Y. Liu, J. Sun, and J. S. Dong, "Symbolic analysis of an electric vehicle charging protocol," in *Proc. 19th Int. Conf. Eng. Complex Comput. Syst.*, 2014, pp. 11–18.
[20] I. Cervesato, N. A. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov, "A meta-notation for protocol analysis," in *Proc. IEEE Workshop Comput. Secur. Found.*, 1999, pp. 55–69.
[21] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *Proc. 25th Int. Conf. Comput. Aided Verification*, 2013, pp. 696–701.
[22] C. Cremers, "The Scyther tool: Verification, falsification, and analysis of security protocols," in *Proc. Int. Conf. Comput. Aided Verification*, 2008, pp. 414–418.
[23] CCITT, "The directory authentication framework - Version 7," 1987, *draft Recommendation X.509*.
[24] M. Abadi and R. M. Needham, "Prudent engineering practice for cryptographic protocols," *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 6–15, Jan. 1996.
[25] 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.
[26] G. Lowe, "An attack on the Needham-Schroeder public-key authentication protocol," *Inf. Process. Lett.*, vol. 56, pp. 131–133, 1995.
[27] T. Chothia, B. Smyth, and C. Staite, "Automatically checking commitment protocols in ProVerif without false attacks," in *Proc. Int. Conf. Principles Secur. Trust*, 2015, pp. 137–155.
[28] H. Krawczyk, "SKEME: A versatile secure key exchange mechanism for internet," in *Proc. Symp. Netw. Distrib. Syst. Secur.*, 1996, pp. 114–127.
[29] SPA tool and experiment models. [Online]. Available: http://lilissun.github.io/r/time.html
[30] MIT, "Kerberos V implementation krb5–1.13," 2014. [Online]. Available: http://web.mit.edu/kerberos/
[31] LDAP Account Manager, "Kerberos V implementation heimdal-1.5.2," 2014. [Online]. Available: http://www.h5l.org
[32] V. Cheval and V. Cortier, "Timing attacks in security protocols: Symbolic framework and proof techniques," in *Proc. 4th Int. Conf. Principles Secur. Trust*, 2015, pp. 280–299.
[33] M. I. Kanovich, T. B. Kirigin, V. Nigam, A. Scedrov, and C. L. Talcott, "Discrete versus dense times in the analysis of cyber-physical security protocols," in *Proc. 4th Int. Conf. Principles Secur. Trust*, 2015, pp. 259–279.
[34] G. Bella and L. C. Paulson, "Kerberos version 4: Inductive analysis of the secrecy goals," in *Proc. 5th Eur. Symp. Res. Comput. Secur.*, 1998, pp. 361–375.
[35] J. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using Murφ;," in *Proc. IEEE Symp. Secur. Privacy*, 1997, pp. 141–151.
[36] J. T. Kohl, B. C. Neuman, and T. Y. T'so, "The evolution of the Kerberos authentication system," in *Distributed Open Systems*. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1994, pp. 78–94.
[37] J. Kohl and B. C. Neuman, *The Kerberos Network Authentication Service (Version 5). Internet Request for Comments RFC-1510*. RFC Editor, 1993.
[38] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad, "Formal analysis of Kerberos 5," *Theoretical Comput. Sci.*, vol. 367, pp. 57–87, 2006.
[39] D. A. Basin and C. J. F. Cremers, "Modeling and analyzing security in the presence of compromising adversaries," in *Proc. 15th Eur. Symp. Res. Comput. Secur.*, 2010, pp. 340–356.
[40] J. D. Guttman, "State and progress in strand spaces: Proving fair exchange," *J. Autom. Reasoning*, vol. 48, no. 2, pp. 159–195, 2012.
[41] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 163–178.
[42] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.
[43] N. A. Lynch and F. W. Vaandrager, "Action transducers and timed automata," *Formal Aspects Comput.*, vol. 8, no. 5, pp. 499–538, 1996.
[44] S. Schneider, *Concurrent and Real Time Systems: The CSP Approach*, 1st ed. New York, NY, USA: Wiley, 1999.
[45] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Int. J. Softw. Tools Technol. Transfer*, vol. 1, no. 1/2, pp. 134–152, 1997.
[46] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *Proc. 10th Int. Conf. Comput. Aided Verification*, 1998, pp. 546–550.
[47] M. Arapinis, J. Liu, E. Ritter, and M. Ryan, "Stateful applied pi calculus," in *Proc. 3rd Int. Conf. Principles Secur. Trust*, 2014, pp. 22–41.
[48] B. Blanchet, M. Abadi, and C. Fournet, "Automated verification of selected equivalences for security protocols," *J. Logic Algebraic Program.*, vol. 75, no. 1, pp. 3–51, 2008.
[49] C. G. Günther, "An identity-based key-exchange protocol," in *Proc. Advances Cryptology Workshop Theory Appl. Cryptographic Techn.*, 1989, pp. 29–37.
[50] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, 1996.



Li Li received the bachelor's degree of information security from the Huazhong University of Science and Technology (HUST), in 2011 and the PhD degree in software engineering and computer security from National University of Singapore (NUS), in 2015. He was a postdoctoral fellow with Singapore University of Technology and Design (SUTD) for a year. He is currently working at SEA as senior software engineer. His research interests focus on security protocol verification, secure system design, and formal methods.

Jun Sun received the bachelor's and PhD degrees in computing science from the National University of Singapore (NUS), in 2002 and 2006, respectively. He is currently an associate professor with Singapore University of Technology and Design (SUTD). In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship. He has been a faculty member of SUTD since 2010. He was a visiting scholar at MIT from 2011-2012. His research interests include software engineering, formal methods, program analysis, and cyber-security. He is the co-founder of the PAT model checker.

Yang Liu received the bachelor of computing from the National University of Singapore (NUS), in 2005 and the PhD degree and continued with his post doctoral work in NUS, in 2010. Since 2012, he joined Nanyang Technological University as an assistant professor. His research focuses on software engineering, formal methods and security. Particularly, he specialises in software verification using model checking techniques. This work led to the development of a state-of-the-art model checker, process analysis toolkit.

Meng Sun received bachelor's and PhD degrees in applied mathematics from Peking University, in 1999 and 2005, respectively. He is currently an associate professor with Peking University. He spent one year as a postdoctoral researcher with National University of Singapore. From 2006 to 2010, he worked as a scientific staff member at CWI. He has been a faculty member of Peking University since 2010. His research interests include software engineering, formal methods, software verification and testing, coalgebra theory, and cyber-physical systems.

Jin-Song Dong received the bachelor's (Hons. I) and PhD degrees in computing from the University of Queensland, in 1992 and 1996. From 1995 to 1998, he was research scientist at CSIRO Australia. Since 1998, he has been in the School of Computing, National University of Singapore (NUS), where he received full professorship in 2016. He is on the editorial board of the *ACM Transaction on Software Engineering and Methodology* and the *Formal Aspects of Computing*.