Automated Verification of Timed Security Protocols with Clock Drift

Li Li[1], Jun Sun[1], and Jin Song Dong[2]

[1] Singapore University of Technology and Design
[2] National University of Singapore

Abstract. Time is frequently used in security protocols to provide better security. For instance, critical credentials often have limited lifetime which improves the security against brute-force attacks. However, it is challenging to correctly use time in protocol design, due to the existence of clock drift in practice. In this work, we develop a systematic method to formally specify as well as automatically verify timed security protocols with clock drift. We first extend the previously proposed *timed applied π-calculus* as a formal specification language for timed protocols with clock drift. Then, we define its formal semantics based on *timed logic rules*, which facilitates efficient verification against various security properties. Clock drift is encoded as parameters in the rules. The verification result shows the constraints associated with clock drift that are required for the security of the protocol, e.g., the maximum drift should be less than some constant. We evaluate our method with multiple timed security protocols. We find a time-related security threat in the TESLA protocol, a complex time-related broadcast protocol for lossy channels, when the clocks used by different protocol participants do not share the same clock rate.

1 Introduction

Time is essential in cyber-security, e.g., message transmissions and user authentications are often required to be finished in a timely manner. In order to check the relevant timing requirements, *timestamps* are constructed from *clocks*, sent through networks and checked by participants in security protocols. For example, in order to deliver a message m timely, the sender first attaches its current clock reading t_s to m and sends them in a secure way. Then, when the receiver obtains t_s and m, it checks t_s against its own clock reading t_r with $t_r - t_s \leq p$ to ensure that m is received within a certain timing threshold p. In the above example, the untimed security (m is not tampered, replayed nor disclosed) and the timed security (m is delivered in time) are equally important. Given a timed protocol, existing literatures [12,16] focus on checking its security when the clocks of different protocol participants are fully synchronized. However, in practice, timestamps are often generated and checked based on different local clocks without perfect synchronization, which could compromise the security proved based on the assumption of perfect clock synchronization. Hence, this work studies the security of timed protocols with the present of the clock drift.

Clock drift commonly exists in practice. For instance, in sensor networks, cheap sensors usually do not have enough resources to maintain accurate clock rate and precise clock reading. Hence, small clock drift should be expected and considered in their

applications. Even though the local clocks can be synchronized at runtime over the network, various unavoidable factors, e.g., network delay, traffic congestion, can lead to a certain level of inaccuracy. Furthermore, when attackers are present in the network, they may attack the clock synchronization protocol [23]. In such a case, the local clocks under the attack may have large clock drift. As a result, when the security depends on the clock reading, the protocol should provide counter-measures for the clock drift.

Clock drift can cause insecurity of timed protocols because the protocol participants rely on local clocks in practice, whereas the security protocol is designed based on the global clock. For instance, in the above message transmission example, let t'_s and t'_r be the readings of the global clock when t_s and t_r are read from the local clocks respectively. The receiver deems the message as timely by checking $t_r - t_s \leq p$. However, the security property requires $t'_r - t'_s \leq p$ to ensure a timely message transmission. In order to capture the inconsistency between local clocks and the (fictional) global clock, we first extend *timed applied π-calculus* [16] to formally specify clock drift in protocol models. Then, we define the semantics of the local clocks in Section 4, which captures their relationship to the global clock. By using this semantics, we can answer the following two security questions. First, our work can check whether a protocol is secure with the presence of clock drift. More importantly, our work can find out how much clock drift can be tolerated in a timed security protocol. We extend SPA, a verification tool we developed in [15,16], with the new calculus and semantics for clock drift. In this work, we use a corrected version [12] of Wide Mouthed Frog [7] as a running example to illustrate our specification and verification method. We apply our method to a number of timed security protocols and successfully find a security threat in TESLA [22,21] in Section 5.2, a complex time-related broadcast protocol for lossy channels, when the clocks used by different protocol participants do not share the same clock rate.

2 Specification

In this section, we first introduce CWMF [12], a *corrected* version of Wide Mouthed Frog [7] protocol, as a running example. When the local clocks of the protocol participants in CWMF are assumed to be perfectly synchronized, CWMF can be verified as secure [12,14] The verification proves that a secret session key can be established among its participants within a certain time. However, it is unclear whether clock drift, which is unavoidable in practice, would compromise the security of CWMF. In the following, we first present CWMF in details and then demonstrate how *timed applied π-calculus*, extended with local clocks, can be used to model such protocols.

2.1 Corrected Wide Mouthed Frog

CWMF is designed to establish a timely fresh session key k from an initiator A to a responder B through a server S. In CWMF, whenever a message is received, the receiver checks the message freshness before accepting it. To be general, we use a parameter p_m to represent the maximum message lifetime. Additionally, we consider the minimal network delay as a parameter p_n. Since p_n is a timing parameter related to

the network environment, it is not directly used in the protocol specification. Instead, it is a compulsory delay that applies to all of the network transmissions.

CWMF is a key exchange protocol that involves three participants: an initiator A, a responder B and a server S. By assumption, A and B have registered their secret long-term keys at the server respectively. The registered key of a user u is written as $key(u)$, which is used to encrypt all network communications between the user and the server. Whenever a message m is transmitted between a user u and the server S, the message m is encrypted by the symmetric encryption function enc_s written as $enc_s(m, key(u))$. CWMF then can be described as the following three steps.

(1) A generates a random session key k at its local time t_a
$A \rightarrow S \quad : \langle A, enc_s(\langle t_a, B, k, tag_1\rangle, key(A))\rangle$
(2) S receives the request from A at its local time t_s
S checks $: t_s - t_a \leq p_m$
$S \rightarrow B \quad : enc_s(\langle t_s, A, k, tag_2\rangle, key(B))$
(3) B receives the message from S at its local time 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 its local time t_a and initiates the CWMF protocol with B by sending its name A and the request $\langle t_a, B, k, tag_1\rangle$ encrypted by $key(A)$ to S. Second, after receiving the request from A at S's local time t_s, S ensures the message freshness by checking $t_s - t_a \leq p_m$. Then, S accepts A's request by forwarding the request $\langle t_s, A, k, tag_2\rangle$ encrypted by $key(B)$ to B. It informs B that S receives a request from A at its local time t_s to communicate with B using the key k. tag_1 and tag_2 are two constants that are used to distinguish these two messages. CWMF uses them to prevent the reflection attack [18] in the original Wide Mouthed Frog protocol [7]. Third, B checks the message freshness again and accepts the request from A if the message is received in a timely fashion. All of the transmitted messages are encrypted under the users' long-term keys that are pre-registered at S.

2.2 Timed Applied π-calculus

Timed applied π-calculus works as a specification language for timed protocols. It is essentially the calculus proposed in [16,2] with the extensions of local clocks and clock drift. Table 1 presents its syntax with the extensions highlighted in the **bold** font.

In *timed applied π-calculus*, we compose *messages* using *functions*, *names*, *nonces*, *variables* and *timestamps*. Functions are 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 consumed timing range. When m is exactly the same as $f(m_1, m_2, \ldots, m_n)$, we call the function a *constructor*; otherwise, it is a *destructor*. For simplicity, we add some syntactic sugar as follows: (1) when $D = [0, \infty)$ which is the largest timing range of functions, we omit '$@\ D$' in the function definition; (2) for constructors, we omit '$\Rightarrow m$' in the definition. For instance, the symmetric encryption function is defined as $enc_s(m, k)$, and its decryption function is defined as $dec_s(enc_s(m, k), k) \Rightarrow m$. Some frequently used functions are defined in Appendix

Type	Expression	
Message(m)	$f(m_1, m_2, ..., m_n)$	(function)
	A, B, C	(name)
	n, k	(nonce)
	t, t_1, t_i, t_n	(timestamp)
	x, y, z	(variable)
Parameter(p)	p, p_1, p_j, p_m	(parameter)
Clock(c)	c, c_1, c_k, c_s	**(clock)**
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)
	$P \vert Q$	(parallel)
	$!P$	(replication)
	$\nu n.P$	(nonce generation)
	$\mu t.P$	(global clock reading)
	$\mu t : c.P$	**(local clock reading)**
	if $m_1 = m_2$ then P [else Q][a]	(untimed condition)
	if B then P [else Q]	(timed condition)
	wait μt until B then P	(global timing delay)
	wait $\mu t : c$ until B then P	**(local timing delay)**
	let $x = f(m_1, \ldots)$ then P	(function application)
	$in(x).P$	(channel input)
	$\overline{out}(m).P$	(channel output)
	check m in 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)

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

Table 1. Syntax of Timed Applied π-Calculus

C. Names are globally shared strings. Nonces are freshly generated random numbers. Variables are memory locations for holding messages. Timestamps are clock readings. Additionally, *parameters* are configurable constants (e.g., the maximum message lifetime p_m) and persistent settings (e.g., the minimal network latency p_n) in the protocol.

In this work, we extend [16] with local clocks. That is, timestamps can be read from these local clocks rather than the shared global clock. For instance, in CWMF, the local clocks of A, S and B can be declared as c_a, c_s and c_b respectively.

The constraint set $B = \mathcal{CS}(t_1, \ldots, t_n, p_1, \ldots, p_m)$ represents a set of linear constraints over timestamps and parameters, which can acts as protocol checking conditions and environment assumptions in the protocol. For instance, given the minimal network latency p_n, when a message sent at t is received at t', we have $t' - t \geq p_n$. Additionally, the configuration $L = \mathcal{CS}(p_1, \ldots, p_m)$ is a set of linear constraints over parameters that

should be satisfied globally. For example, the configuration $p_n > 0$ should be satisfied because the message transmission delay should always be positive.

As shown in Table 1, processes are defined as follows. '0' is a null process that does nothing. '$P|Q$' is a parallel composition of processes P and Q. The replication '$!P$' stands for an infinite parallel composition of process P, which captures an unbounded number of protocol sessions running in parallel. The nonce generation process '$\nu n.P$' represents that a fresh nonce n is generated and bound to process P. The global clock reading process '$\mu t.P$' means that a timestamp t is read from the global clock and bound to process P. The local clock reading process '$\mu t : c.P$' similarly means that a timestamp t is read from a local clock c and bound to process P. The checking condition $cond$ in the 'if $cond$ then P else Q' process has two forms: 1) the untimed condition $m_1 = m_2$ is a symbolic equivalence checking between two messages; 2) the timed condition $\mathcal{CS}(t_1, t_2, \ldots, t_n, p_1, p_2, \ldots, p_m)$ is a constraint over timestamps and parameters. When $cond$ evaluates to true, process P is executed; otherwise, Q is executed. The global timing delay process 'wait μt until B then P' means that P is executed until the reading t from the global clock satisfies the timing condition B. Similarly, the local timing delay process 'wait $\mu t : c$ until B then P' means that P is executed until the reading t from a local clock c satisfies the timing condition B. The function application 'let $x = f(m_1, \ldots, m_n)$ then P' 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. The channel input '$in(x).P$' means that a message, bound to the variable x, should be received before executing P. The channel output '$\overline{out}(m).P$' describes that the message m shall be sent out before executing process P. The uniqueness checking expression 'check m in db as unique then P' ensures that (1) the value of m does not exist in a database db before this expression, and (2) m is inserted into db after this expression. The uniqueness checking is particularly useful for preventing replay attacks in practice.

Additionally, the *init*, *join* and *accept* events are introduced to specify the security properties. They represent the initialization, participation and acceptance of the protocol participants respectively according to their roles, which are elaborated in Section 3.

Notations and Definitions. For simplicity, $tuple_n(m_1, m_2, \ldots, m_n)$ is simply written as $\langle m_1, m_2, \ldots, m_n \rangle$. 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 $\langle x_1, x_2, \ldots, x_n \rangle$. When we only want to check that a variable x_i equals to a constant C, we can replace 'x_i' with '$=C$' in the above tuple.

Remarks. We do not need special syntax to specify private channels in *timed applied π-calculus*. Private channels can be constructed with public channels and unbreakable encryptions. For instance, in order to model a message m transmitted in a private channel, we first introduce a secret key k_s. Then, we can model a private channel as $\overline{out}(enc_s(m, k_s)).P | in(x).$let $m' = dec_s(x, k_s)$ then Q.

2.3 CWMF Model

In order to verify CWMF in a hostile environment, we make the following assumptions. (1) The adversary can ask any protocol participant to join the protocol, including A, S and B. (2) The adversary controls the protocol participation time, e.g., the initialization time of A in CWMF. (3) S provides its session key exchange service to all of its registered users. (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 CWMF, because we are interested in the protocol acceptance between legitimate users, we assume that B only accepts requests from A. Additionally, a public channel controlled by the adversary is used in CWMF 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 (possibly personated by the adversary), which can be modeled as the process P_r below.

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

In CWMF, A takes a role of the initiator as specified by P_a below. It first starts the protocol by receiving a responder's name r from c, assuming that r is specified by the adversary. Then, A generates a session key k and reads t_a from its local clock c_a. Then, A emits an $init$ event to indicate the protocol initialization with the arguments A, r, k at t_a. Finally, the message $\langle A, enc_s(\langle t_a, r, k, tag_1\rangle, key(A))\rangle$ is sent from A to S.

$$P_a \triangleq in(r).\nu k.\mu t_a : c_a.init(A, r, k)@t_a.\overline{out}(\langle A, enc_s(\langle t_a, r, k, tag_1\rangle, key(A))\rangle).0$$

As specified by the process P_s, after S receives a user's request as a tuple $\langle i, x\rangle$, it records its local time from c_s as t_s and decrypts x using $key(i)$. If the decryption is successful, it obtains the initialization time t_i, the responder's name r and the session key k. When the freshness checking $t_s - t_i \leq p_m$ is passed, S then believes that it is participating in a protocol run at time t_s and engages the $join$ event. Later, a new message encrypted by the responder's key, written as $enc_s(\langle t_s, i, k, tag_2\rangle, key(r))$, is sent to the responder over the public channel.

$$\begin{aligned} P_s \triangleq\ & in(\langle i, x\rangle).\mu t_s : c_s.\mathit{let}\ \langle t_i, r, k, {=}tag_1\rangle = dec_s(x, key(i))\ \mathit{then} \\ & \mathit{if}\ t_s - t_i \leq p_m\ \mathit{then}\ join(i, r, k)@t_s.\overline{out}(enc_s(\langle t_s, i, k, tag_2\rangle, key(r))).0 \end{aligned}$$

Additionally, as shown in the process P_b, when B receives the message from S, B records its local 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 $2 * p_m$ and engages the accept event at time t_b.

$$\begin{aligned} P_b \triangleq\ & c(x).\mu t_b : c_b.\mathit{let}\ \langle t_s, {=}A, k, {=}tag_2\rangle = dec_s(x, key(B))\ \mathit{then} \\ & \mathit{if}\ t_b - t_s \leq p_m\ \mathit{then}\ \mathit{check}\ k\ \mathit{in}\ db\ \mathit{as}\ \mathit{unique}\ \mathit{then}\ accept(A, B, k)@t_b.0 \end{aligned}$$

Finally, we have a process $P_p \triangleq \overline{c}(A).\overline{c}(B).0$ that broadcasts the names A and B. The overall process $P \triangleq (!P_r)|(!P_a)|(!P_s)|(!P_b)|(!P_p)$ is a parallel composition of the infinite replications of the five processes described above.

3 Timed Security Properties

In this section, we define the timed security properties. Notice that the properties are defined based on the global clocks, whereas the participants in the protocols rely on local clocks in practice. In this work, we focus on the authentication properties, as they can be largely affected by clock drift. We first introduce the adversary model as follows.

Adversary Model. We assume that an active attacker exists in the network, whose capabilities are extended from the Dolev-Yao model [13]. The attacker can intercept all communications, compute new messages, generate new nonces and send the messages he obtained. Additionally, he can use all the publicly available functions, e.g., encryption, decryption, concatenation. He can also ask the genuine protocol participants to take part in the protocol at any time. Comparing our attack model with the Dolev-Yao model, reading timestamps from various clocks, attacking weak cryptographic functions and compromising legitimate protocol participants are allowed additionally. A formal definition of the adversary model is defined as follows.

Definition 1. ***Adversary Process.*** *The adversary is defined as an arbitrary* closed *timed applied π-calculus process K which does not emit the* init*,* join *and* accept *events.*

Timed Authentication. In a protocol, we often have an initiator who starts the protocol and a responder who accepts the protocol. For instance, in CWMF, A is the initiator and B is the responder. Additionally, other entities called partners, e.g., S in CWMF, can be involved during the protocol execution. In general, the protocol authentication aims at establishing common knowledge among the protocol participants when the protocol successfully ends. Specifically, for timed protocols, the common knowledge contains the information on the participants' time.

Since different participants take different roles in the protocol, we introduce the *init*, *accept* and *join* events for the initiator, the responder and the partners respectively. Whenever a protocol participant believes that it is participating in a protocol as a certain role, it engages the corresponding event with the protocol parameters and the correct time. For instance, in CWMF, A engages $init(A, r, k)@t_a$; S engages $join(i, r, k)@t_s$; and B engages $accept(i, B, k)@t_b$. We remark that t_a, t_s and t_b in above events should be the correct readings from the global clock, which could be different from the values used for constructing messages in the protocol.

Based on the *init*, *join* and *accept* events, the protocol authentication properties then can be formally specified as event correspondences. The timed non-injective authentication is satisfied if and only if for every acceptance of the protocol responder, the protocol initiator indeed initiates the protocol and the protocol partners indeed join in the protocol, agreeing on the protocol arguments and timing requirements. We formally define the non-injective timed authentication as follows.

Definition 2. ***Non-injective Timed Authentication.*** *The non-injective timed authentication, denoted as $Q_n = accept \leftarrow\!\!\lfloor\ B\ \rfloor\!\!- init, join_1, \ldots, join_n$, 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 and the timing constraints B.*

In CWMF, the non-injective timed authentication can be written as

$$Q_n = accept(i,r,k)@t_r \\ \leftarrow\!\!\lfloor\ t_s - t_i \leq \S p_m \wedge t_r - t_s \leq \S p_m\ \rfloor\!\!\rightarrow \\ init(i,r,k)@t_i, join(i,r,k)@t_s.$$

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

Definition 3. ***Injective Timed Authentication.*** *The injective timed authentication, denoted as $Q_i = accept \leftarrow\!\!\lfloor\ B\ \rfloor\!\!\rightarrow init, join_1, \ldots, join_n$, is satisfied by a closed process P, if and only if, (1) the non-injective timed authentication $Q_n = accept \leftarrow\!\!\lfloor\ B\ \rfloor\!\!\rightarrow init, join_1, \ldots, join_n$, 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 B in global time.*

For simplicity, given a non-injective authentication property $Q_n = accept \leftarrow\!\!\lfloor\ B\ \rfloor\!\!\rightarrow H$ and its injective version $Q_i = accept \leftarrow\!\!\lfloor\ B\ \rfloor\!\!\rightarrow H$, we define two functions such that $inj(Q_n) = Q_i$ and $non\_inj(Q_i) = Q_n$. Hence, we can write injective timed authentication of CWMF as $Q_i = inj(Q_n)$.

4 Semantics of Clock Drift

In this section, we first briefly introduce the *timed logic rules* [16] which are used to capture the semantics of the *timed applied π-calculus*. We use CWMF to demonstrate how *timed logic rules* can be used to capture the semantics of *timed applied π-calculus*. Particularly, we capture the semantics of reading timestamps from local clocks based on two different ways of modeling clock drift. We use these two different semantics to show that our method can be adopted to handle different scenarios in practice. We have implemented these two different clock drift semantics in SPA [16].

4.1 Timed Logic Rules

In [16], we proposed the *timed logic rules* to define the semantics of the *timed applied π-calculus* in terms of the adversary capabilities, so timed security protocols can be verified efficiently. In this work, we show how to use them to capture clock drift. When the semantics of calculus processes are represented by logic rules, we need additional notations to differentiate the data types of names, nonces, timestamps, variables and parameters as shown in Table 2. (1) The syntax of variables and functions are unchanged. (2) Names are appended with a pair of square brackets from A to $A[]$. (3) Nonces are put inside of a pair of square brackets from n to $[n]$. (4) Timestamps are written with a blackboard bold font from t to $\mathbb{t}$. (5) Parameters are prefixed from p to $\S p$.

Generally, each timed logic rule specifies a capability of the adversary in the form of $[\ G\ ]\ e_1, e_2, \ldots, e_n \dashv\!\!\lfloor\ B\ \rfloor\!\!\rightarrow e$. G is a set of untimed guards, $\{e_1, e_2, \ldots, e_n\}$ is a

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

Table 2. Syntax of Timed Logic Rules

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 e is ready to occur. When G is empty, we simply omit '$[G]$' in the rule.

The events represent the things that can occur in the protocol. In this work, six types of events are essential to the timed protocols with clock drift. Similar to the *timed applied π-calculus*, we have event $init$, $join$ and $accept$ that signal the authentication claims made by the legitimate protocol participants. In particular, the $init$, $join$ events appear in the premise part whereas the $accept$ events appear in the conclusion part. We amend the events from $init(m)@t$, $join(m)@t$ and $accept(m)@t$ to $init([d], m, \mathbb{t})$, $join([d], m, \mathbb{t})$ and $accept([d], m, \mathbb{t})$ respectively. The additional nonce $[d]$ represents the session id, which is specifically introduced to check the authentication properties.

Additionally, $know(m, \mathbb{t})$ means that the adversary obtains message m at time $\mathbb{t}$. Because the adversary intercepts all communications over the public channel, for every network input $in(x)$ at time t, we add $know(x, \mathbb{t}')$ satisfying $\mathbb{t}' \leq \mathbb{t}$ to the rule premises, meaning that the adversary need to know x before time t so as to send it at t; for every network output $\overline{out}(m)$ at time t, we construct a rule that concludes $know(m, \mathbb{t}')$ and satisfies $\mathbb{t}' - \mathbb{t} \geq \S p_n$, representing m can be intercepted by the adversary after the network delay $\S p_n$. Furthermore, given a nonce generated in $\nu n.P$, we add $new([n], l[])$ to the rule premises, denoting the generation of nonce $[n]$ at the process location $l[]$ (we use unique labels to represent different locations in the process). Lastly, $unique(u, db[], m)$ means that the message u should have a unique value in a database $db[]$ (any constant can be a database name). Given the above unique event constructed in a process, m is an ordered tuple of messages that can be identified by $\langle u, db[]\rangle$, consisting of the

network inputs, generated nonces and read timestamps in the chronological order until the process ends or its sub-process is an infinite replication process. Unique events and new events are constructed in the following two cases: (1) when 'check u in db as unique then P' is present in the process, $unique(u, db[], m)$ is added; (2) given '$\nu n.P$' in the process at the location l, $new(n, l[])$ and $unique(n, l[], m)$ are added. The location names are generated by a special function $loc()$, which returns a unique name to represent the current process location. The semantics of *timed applied π-calculus* is presented in Appendix A.

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 of the same message; $unique$ events with the same unique value and database; $init$, $join$ or $accept$ events with the same session id. In general, each event is associated with a signature and premise events with the same signatures in a rule should be merged. As shown in Table 2, event signature can be constructed by concatenating its event name with a sequence of messages prefixed by '$\star$'. For instance, in the event $unique(\star u, \star l[], m)$, the unique value u and the location $l[]$ is prefixed by $\star$, so its signature is '$unique.u.l[]$', where '.' concatenates and separates the strings.

To provide a better understanding of the timed logic rules, we show three examples where clock drift does not exist in the following. Later, we compare them with those rules with clock drift.

Example 1. 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 needs to know a message m and a key k. Then, the encryption function returns the encrypted message $enc_s(m, k)$. Hence, the encryption can be represented as the following rule.

$$know(m, \mathbb{t}_1), know(k, \mathbb{t}_2) \dashv[\ \mathbb{t}_1 \leq \mathbb{t} \wedge \mathbb{t}_2 \leq \mathbb{t}\ ]\mapsto know(enc_s(m, k), \mathbb{t}) \qquad (1)$$

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 2. In CWMF, the server provides its key registration service to the public as P_s. This service can be captured as follows.

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

It means that anyone can register at the server using any name except A and B. □

Example 3. In this example, we demonstrate the *timed logic rule* for P_b in CWMF, when B reads the timestamps from the global clock rather than its local clock. B receives a message $enc_s(\langle t_s, A, k, tag_2\rangle, key(B))$ from S, records its current time as t_b and claims acceptance if $t_b - t_s \leq p_m$. Since the adversary can start the protocol at anytime, we assume that t_b is specified by the adversary. Then, the *timed logic rule* of P_b is written as the following rule, where $m_b = \langle enc_s(\langle \mathbb{t}_s, A[], k, tag_2[]\rangle, key(B[])), \mathbb{t}_b\rangle$.

$$\begin{aligned} &unique(k, db[], m_b), new([n_b], l_b[]), unique([n_b], l_b[], m_b) \\ &\quad, know(\mathbb{t}_b, \mathbb{t}_b), know(enc_s(\langle \mathbb{t}_s, A[], k, tag_2[]\rangle, key(B[])), \mathbb{t}_1) \\ &\qquad \dashv[\ \mathbb{t}_1 \leq \mathbb{t}_b \wedge \mathbb{t}_b - \mathbb{t}_s \leq \S p_m\ ]\mapsto accept([n_b], \langle A[], B[], k\rangle, \mathbb{t}_b) \end{aligned} \qquad (2)$$

In Section 4.2, we will compare it with the rules explicitly modeling the clock drift. □

4.2 Semantics of Local Clocks

In this work, we additionally introduce the operation $\mu t : c$ that reads a timestamp t from a local clock c. This operation is applicable to the local clock reading process and the local timing delay process shown in Table 1. In order to capture the semantics of timestamps constructed with $\mu t : c$ in the calculus, we need to record two timestamps $\mathbb{t}$ and $\mathbb{t}_g$ from the local clock c and the global clock respectively. The semantics of regular operations in protocol execution, e.g., message constructions and guard conditions, is defined based on the local time $\mathbb{t}$ because they use the real values read from local clocks. However, the semantics of the security claims, i.e., $init$, $join$ and $accept$ events, should be defined based on the global time $\mathbb{t}_g$ to indicate the correct timing of event engagement. In this way, we can correctly specify and distinguish two different types of timestamps that are (1) used in the protocol execution and (2) captured by the security properties. Hence, the remaining task is to establish the relation between $\mathbb{t}$ and $\mathbb{t}_g$ based on the assumptions of the clock drift. In the following, we show two different ways of modeling clock drift. Notice that, when all of the timestamps are read from the global clock, the *timed logic rules* remain the same as those in [16]. For instance, the *timed logic rules* in Example 1 and Example 2 remain the same, while the *timed logic rule* in Example 3 shall be updated to take clock drift into account. In this work, we consider two different scenarios of clock drift: (**VR**) different clocks have different clock rates but concern their maximum drift bounds; (**SR**) different clocks share the same clock rate but have different readings. The differences between VR and SR in the following *time logic rules* are highlighted in the red font.

Variable Clock Rate (VR). In VR, we assume that the local clock rate can vary during the protocol execution. That is, local clocks can run faster or slower than the global clock from time to time. Additionally, we assume that their maximum clock drift are bounded, resulting in the following two properties. First, the timestamps read from the same local clock should always be monotonic. For example, given a process $\mu t_1 : c.\mu t_2 : c.0$, we have $t_1 \leq t_2$. However, if t_1 and t_2 are read from two different local clocks, e.g., $\mu t_1 : c_1.\mu t_2 : c_2.0$, t_2 could be smaller than t_1. Second, the differences between a local clock and the global clock are always bounded by a maximum drift parameter associated with that local clock. For instance, given a timestamp t read from c at global time t', we have $|t - t'| \leq p_c$, where p_c is the maximum drift of c, satisfying $p_c > 0$. If VR is assumed, the *timed logic rule* of P_b can be written as the following rule, where $m'_b = \langle enc_s(\langle \mathbb{t}_s, A[], k, tag_2[]\rangle, key(B[])), \langle \mathbb{t}_b, \mathbb{t}'_b\rangle\rangle$.

$$unique(k, db[], m'_b), new([n_b], l_b[]), unique([n_b], l_b[], m'_b)$$
$$, know(\mathbb{t}_b, \mathbb{t}'_b), know(enc_s(\langle \mathbb{t}_s, A[], k, tag_2[]\rangle, key(B[])), \mathbb{t}_1)$$
$$-[\ \mathbb{t}_1 \leq \mathbb{t}'_b \wedge \mathbb{t}_b - \mathbb{t}_s \leq \S p_m \wedge |\mathbb{t}'_b - \mathbb{t}_b| \leq \S p_b\ ]\mapsto accept([n_b], \langle A[], B[], k\rangle, \mathbb{t}'_b)$$

Shared Clock Rate (SR). When the local clocks share the same clock rate of the (correct) global clock, the differences of the readings from different clocks are always the same. In this case, we introduce a clock drift parameter d_c for each clock c. Whenever a

timestamp t is read from c at the global time t', we have $t = t' + d_c$. Hence, in this case, given the two timestamps extracted from the same local clock, their difference reflects the exact duration of that time period. For instance, the *timed logic rule* of P_b can be written the following rule, where d_b is the clock drift of c_b and m'_b is the same as above.

$$\begin{aligned} &unique(k, db[], m'_b), new([n_b], l_b[]), unique([n_b], l_b[], m'_b) \\ &\quad, know(\mathbb{t}_b, \mathbb{t}'_b), know(enc_s(\langle \mathbb{t}_s, A[], k, tag_2[] \rangle, key(B[])), \mathbb{t}_1) \\ &\qquad \dashv\!\lfloor\, \mathbb{t}_1 \leq \mathbb{t}'_b \wedge \mathbb{t}_b - \mathbb{t}_s \leq \S p_m \wedge \mathbb{t}_b - \mathbb{t}'_b = \S d_b \,\rfloor\!\vdash\!\!\mapsto accept([n_b], \langle A[], B[], k \rangle, \mathbb{t}'_b) \end{aligned}$$

Comparing VR and SR. The difference between VR and SR can be illustrated with the calculation of the round-trip delay (RTD) in the Network Time Protocol (NTP). NTP is designed to synchronize the clocks between a client A and a server B. In NTP, A first reads its clock c_a as t_a and then sends an authenticated signal to B. Once B receives the signal, it reads its clock c_b as t_b. After B verifies the signal successfully, B reads its clock c_b as t'_b and replies another authenticated signal back to A. Once A receives the reply signal, it reads its clock c_a as t'_a. If the reply signal is correctly verified, A calculates the RTD as $\delta = (t'_a - t_a) - (t'_b - t_b)$. When SR is assumed, the calculation of δ is accurate even if clock drift exists. However, when VR is assumed, δ is *not* accurate because the distance of clock drift can vary during the protocol execution.

4.3 Verification Overview

After obtaining the initial *timed logic rules* from the *timed applied π-calculus* as shown above, the security properties then can be verified using the method proposed in [16]. We briefly introduce the method in the following and refer the readers to [16] for details.

In general, the verification method works by composing all of the existing timed logic rules into new rules, by unifying the conclusion of one rule with the premises of other rules. For instance, we can compose Rule (1) to Rule (2) as the following rule.

$$\begin{aligned} &unique(k, db[], m'_b), new([n_b], l_b[]), unique([n_b], l_b[], m'_b) \\ &\quad, know(\mathbb{t}_b, \mathbb{t}_b), know(\langle \mathbb{t}_s, A[], k, tag_2[] \rangle, \mathbb{t}_1), know(key(B[]), \mathbb{t}_2) \\ &\qquad \dashv\!\lfloor\, \mathbb{t}_1 \leq \mathbb{t}_b \wedge \mathbb{t}_2 \leq \mathbb{t}_b \wedge \mathbb{t}_b - \mathbb{t}_s \leq \S p_m \,\rfloor\!\vdash\!\!\mapsto accept([n_b], \langle A[], B[], k \rangle, \mathbb{t}_b) \end{aligned}$$

We repeatedly generate new rules until no new rule can be generated. Then, we use the set of all rules to check the authentication properties, ensuring that no violating rule exists and every authentication property is satisfied. When the above two criteria can be met, the result of the verification is a set of configurations (each configuration is a set of constraints over the parameters). We prove that the protocol is guaranteed to satisfy the security property if its parameters choose values from the configurations. Due to the limitation of space, we demonstrate the full verification process of CWMF in Appendix D. Notice that the verification process is not guaranteed to terminate in general. However, it has been shown that it often terminates for practical protocols [5,14,15]. After obtaining the secure configurations, we need to additionally ensure that clock drift parameters are not constrained by other protocol parameters. If any clock drift parameter is related to other protocol parameters, we denote that the protocol has security **threat**

Protocol	$\sharp\mathcal{R}^a$	No Clock Drift		Shared Clock Rate		Variable Clock Rate	
		Result	Time	Result	Time	Result	Time
Corrected WMF [7,18,16]	80	Secure	47.51ms	Threat	112.75ms	Attack	150.09ms
TESLA [22,21]	343	Secure	3.17s	Threat	3.55s	Threat	4.37s
Auth Range [6,8]	53	Secure	38.58ms	Secure	60.73ms	Attack	46.47ms
CCITT X.509 (1c) [3]	135	Secure	162.69ms	Secure	231.86ms	Secure	224.00ms
CCITT X.509 (3) BAN [7]	198	Secure	791.00ms	Secure	1058.05ms	Secure	969.97ms
NS PK Time [20,17,10]	173	Secure	170.00ms	Threat	205.93ms	Threat	353.20ms

Table 3. Experiment Results

[a] The number of rules generated in the verification.

under the clock drift because those constraints must be checked at runtime in the real application scenarios. For instance, given the network latency p_n and the maximum drift p_c for a local clock c, if $p_c < p_n$ is required in the secure configuration but it cannot be satisfied in the real application scenario, the protocol is vulnerable.

5 Evaluations

Our method has been integrated into the tool named Security Protocol Analyzer (SPA). SPA relies on PPL [4] to check the satisfaction of timing constraints, i.e., to tell whether a set of timing constraints is empty or not. We use SPA to check multiple timed protocols as shown in Table 3. All the experiments are conducted using a Mac OS X 10.10.5 with 2.3 GHz Intel Core i5 and 16G 1333MHz DDR3. In order to clearly demonstrate how clock drift can affect the security of protocols, all of the protocols evaluated in this section are correct under perfect synchronization. The evaluated protocols are *corrected* WMF [7,12], TESLA [22,21], a distance bounding protocol [6,8], *corrected* CCITT [9,3,7]. and a timing commitment version [10,15] of Needham-Schroeder [20,17]. All of the protocols can be verified or falsified for an unbounded number of protocol sessions. SPA and the protocol models are available at [1]. Notice that the security (secure constraints over parameters) is proved 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 found a new clock drift related security threat in TESLA. In the following, we illustrate how SPA works with our running example first and then other protocols.

5.1 CWMF Protocol

Based on the specification of CWMF in Section 2.3, WMF is checked in three different scenarios of clock drift. Let d_a, d_s and d_b be the drift distances of c_a, c_s and c_b respectively.

– When all clocks are perfectly synchronized, in order to finish CWMF, SPA returns that the minimum network latency p_n should be smaller than the maximum message lifetime p_m.

– (SR) When the local clocks share the same clock rate, CWMF is correct if and only if the following constraints are met: (1) $0 \leq d_s - d_a$; (2) $0 \leq d_b - d_s$; (3) $d_s - d_a \leq p_m - p_n$; (4) $d_b - d_s \leq p_m - p_n$. Constraint (1) and (2) ensure that the injective authentication is finished within p_m. Constraint (3) and (4) are required to finish the protocol. Since d_a, d_s and d_b exist in the constraints, which might not be satisfied in practice, the verification result presents a security threat of CWMF.
– (VR) When different local clocks have different clock rates, the constraint returned by SPA is $false$. It means that SPA cannot find the right parameter values to make CWMF secure in the case of VR. Intuitively, the authentication property requires CWMF to be finished within $2 \times p_m$, whereas the protocol itself can only achieve the timing threshold $2 \times p_m + p_a + p_b$. In order to ensure $2 \times p_m + p_a + p_b \leq 2 \times p_m$, we have $p_a + p_b \leq 0$. Since p_a and p_b are positives, SPA cannot find any suitable constraint for these parameters.

5.2 TESLA Protocol

TESLA [22,21] is short for Timed, Efficient, Streaming, Loss-tolerant Authentication protocol. It can provide efficient authenticated broadcast over lossy channels. Generally, it consists of many resource constrained receivers and a relatively powerful sender.

Protocol Description. The goal of TESLA is to transfer a set of messages $\{M_j \mid j \in [0 \ldots n]\}$ from a sender *S* to a receiver *R* in an authenticated manner. Since *R* have limited computing power, *S* cannot adopt signature for authentication purpose because of the large computing overhead. As a result, *S* computes hash values for messages with hash keys and uses these keys for authentication. Specifically, *S* divides the message transmission time into several continuous intervals. Each interval has the same length of p_d ($p_d > 0$). Then, *S* sends the messages with their hash values in different time intervals and reveals the corresponding hash keys in later time intervals. For example, *S* sends $\langle M_j, mac(M_j, k_i)\rangle$ in the i-th time interval and reveals the key k_i in the next interval. Since only *S* knows k_i before k_i is revealed, when k_i is check to be a hash key from *S*, $\langle M_j, mac(M_j, k_i)\rangle$ should be sent from *S*. In order to check the authenticity of the hash keys, TESLA requires these keys to form a chain such that k_i can be computed by k_{i+1} with a one-way function. Hence, when *S* can authenticate the first key k_0 to *R*, *R* can use k_0 to authenticate newly received hash keys. Additionally, using this method, even if some hash key k_i is lost, once k_{i+x} ($x > 0$) is received by *R*, k_i can be computed from k_{i+x} for authentication. In order to provide sound security, *S* in TESLA does not send the hash keys directly. Instead, it sends the hash key generators $\{k'_i\}$ and uses the generators to compute the actual hash keys $\{k_i\}$.

Unlike WMF and many other protocols, TESLA does not assume perfect clock synchronization. It rather requires loose time synchronization between *S* and *R*, where *R* knows the upper-bound of the local clock drift δ between *S* and *R*. In order to obtain the upper-bound, TESLA adopts the following two-step protocol. Firstly, *R* reads its current time as t_r, generates a nonce (a random number) n and sends n to *S*. Secondly, *S* reads its current time as t_s, sign t_s and n with its private signing key sk_s and sends the signature back to *R*. When *R* receives the signature from *S*, *R* can be sure that δ has an upper-bound of $t_s - t_r$. Thereafter, when *R* receives a message from *S* at its local

time t'_r, he can claim that the current time of *S* is upper-bounded by $t'_r + t_s - t_r$. Due to the limited space, the modeling details of TESLA are available in Appendix B.

Verification Results. When TESLA is checked with *SR* or no clock drift, it is verified as correct with the requirement $2 \times p_n < p_d$, i.e., the length of every interval p_d should be larger than twice of the minimal network latency p_n. To the best of our knowledge, this configuration requirement, justified in the following, has not be reported in any other literature before. According to the verification result from SPA, this protocol configuration requirement is necessary because of the over-approximation of *S*'s clock at *R*'s side in TESLA. When a payload is sent by S at t'_s and received by R at t'_r based on their local clocks respectively, the *clock synchronization* ensures that $t'_s < t_s^{bound} = t'_r + t_s - t_r$. Additionally, in order to receive and check the payload successfully, t'_s and t_s^{bound} should belong to the same interval. Hence, given an initial time t_0 and an interval index i, we have $t_0 + i \times p_d \leq t'_s < t'_r + t_s - t_r < t_0 + (i+1) \times p_d$, which implies that p_d should be larger than $(t'_r - t_r) - (t'_s - t_s)$. That is, $2 \times p_n < p_d$.

When TESLA is checked with *VR*, SPA automatically reports a new security requirement such that $p_r + p_s \leq p_n$, where p_r and p_s are the maximum clock drift of R and S respectively. This configuration requirement is necessary because the *clock synchronization* alone fails to guarantee the bounding $t'_s < t'_r + t_s - t_r$[3]. Hence, in order to prevent the adversary from using the published keys to construct legal payloads, the sum of the clock drift values from R and S should be smaller than the network latency. This new configuration largely limits the application of TESLA protocol when *VR* is assumed, which is also unreported in existing literatures.

6 Related Works

This work builds on our previous works [14,15]. In this work, we extend the *timed applied π-calculus* with local clocks and clock drift. In order to verify the protocols specified in *timed applied π-calculus*, we define its semantics based on the *timed logic rules* [14,15]. We introduce two clock drift scenarios based on whether the clock rate is shared or not. During the evaluation, we show that our framework is able to verify timed security protocols with clock drift automatically, which is unique comparing with other existing works. The analyzing framework closest to ours was proposed by Delzanno and Ganty [12] which applies $MSR(\mathcal{L})$ to specify unbounded crypto protocols by combining first order multiset rewriting rules and linear constraints. According to [12], 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. Our method can be applied to verify protocols without any protocol modification. Many tools [5,11,19] for verifying *untimed* security protocols are related.

[3] $2 \times p_n < p_d$ in SR has been updated to $2 \times p_n < p_d + 2 \times (p_s + p_r)$ in VR.

7 Conclusions

In this work, we develop a systematic method to formally specify as well as automatically verify timed security protocols with clock drift. We have integrated our method into SPA and used it to analyze several timed protocols. In the experiments, we have found new security threats related to clock drift in TESLA. Since the problem of verifying security protocols is undecidable in general, we cannot guarantee the termination of our method. However, similar to existing works on verifying security protocols [5,14,15], it has been shown that it often terminates for practical protocols.

References

1. SPA tool and experiment models. Available at `http://lilissun.github.io/r/drift.html`.
2. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115, 2001.
3. M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.
4. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *SAS*, pages 213–229. Springer, 2002.
5. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW*, pages 82–96. IEEE CS, 2001.
6. S. Brands and D. Chaum. Distance-bounding protocols (extended abstract). In *EUROCRYPT*, volume 765 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 1993.
7. M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.
8. S. Capkun and J.-P. Hubaux. Secure positioning in wireless networks. *IEEE Journal on Selected Areas in Communications*, 24(2):221–232, 2006.
9. CCITT. The directory authentication framework - Version 7, 1987. Draft Recommendation X.509.
10. T. Chothia, B. Smyth, and C. Staite. Automatically checking commitment protocols in proverif without false attacks. In *POST*, pages 137–155, 2015.
11. C. Cremers. The Scyther tool: Verification, falsification, and analysis of security protocols. In *CAV*, pages 414–418. Springer, 2008.
12. G. Delzanno and P. Ganty. Automatic verification of time sensitive cryptographic protocols. In *TACAS*, pages 342–356. Springer, 2004.
13. D. Dolev and A. C.-C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.
14. L. Li, J. Sun, Y. Liu, and J. S. Dong. Tauth: Verifying timed security protocols. In *ICFEM*, pages 300–315. Springer, 2014.
15. L. Li, J. Sun, Y. Liu, and J. S. Dong. Verifying parameterized timed security protocols. In *FM*, page 342–359. Springer, 2015.
16. L. Li, J. Sun, Y. Liu, M. Sun, and J. S. Dong. A formal specification and verification framework for timed security protocols. Technical report, Singapore University of Technology and Design, 2016.
17. G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.
18. G. Lowe. A family of attacks upon authentication protocols. Technical report, Department of Mathematics and Computer Science, University of Leicester, 1997.

19. S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The Tamarin prover for the symbolic analysis of security protocols. In *CAV*, pages 696–701. Springer, 2013.
20. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.
21. A. Perrig, R. Canetti, D. X. Song, and J. D. Tygar. Efficient and secure source authentication for multicast. In *NDSS*, 2001.
22. A. Perrig, R. Canetti, J. D. Tygar, and D. X. Song. Efficient authentication and signing of multicast streams over lossy channels. In *S&P*, pages 56–73, 2000.
23. K. Sun, P. Ning, and C. Wang. Secure and resilient clock synchronization in wireless sensor networks. *IEEE Journal on Selected Areas in Communications*, 24(2):395–408, 2006.

Appendix A: Semantics of Timed Applied π-calculus

In order to facilitate efficient protocol verification, 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.

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

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

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

- C maps a clock c to its most recently reading $\mathbb{t}_c$ before the execution of P. We use it to maintain a chronological order of all generated timestamps, i.e., for any new timestamp $\mathbb{t}$ from c, we have $\mathbb{t}_c \leq \mathbb{t}$.
- T maps a local clock reading t to its corresponding global clock reading t'.
- f is a variable represents the full execution trace of the current process replication including P, consisting of network inputs, read timestamps and generated nonces in a chronological order. r is a variable represents the rest of the execution trace from P. Since the process is deterministic, the process outputs are excluded from f and l. Their structure is in the form of $\langle m_1, \langle m_2, m_2' \rangle \rangle$, where all of the first message in the tuples $\langle m_1, m_2 \rangle$ form the execution trace. Using this method, we can add another message m_3 to the trace with a substitution $m_2' \mapsto \langle m_3, m_3' \rangle$.
- G is a set of untimed guards that leads to P.
- B is a set of timing constraints that leads to P.
- σ is a substitution that is applicable to P.

Given a process P and its contexts $\langle C, T, f, r, G, H, B, \sigma \rangle$, the timed logic rules extracted from P can be denoted as $\lfloor P \rfloor CTfrGHB\sigma$. 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 2.2, the specification and verification are actually based on '$\mu t_0.\mu t_1 : c_1 \ldots \mu t_n : c_n.!P_0$', where t_0 is the starting time of the whole process and $\forall i \in [1..n]$ we have t_i as the initial timestamp of the local clock c_i. Then, the semantic rule generation can be fired as $\lfloor P_0 \rfloor C_0 \emptyset fr \emptyset \emptyset \mathbb{U} \emptyset$, where f and l are variables, C_0 maps the global clock to t_0 and c_i to t_i.

First, we discuss three types of processes that either terminate the current session or fork sub-sessions. They are the null process '0', the parallel composition process '$P|Q$'

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

$$
\begin{aligned}
&\lfloor 0 \rfloor CTfrGHB\sigma = \emptyset \\
&\lfloor P|Q \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfr_P GHB(\sigma \cdot [r \mapsto \langle r_P, r_Q \rangle]) \\
&\quad \cup \lfloor Q \rfloor CTfr_Q GHB(\sigma \cdot [r \mapsto \langle r_P, r_Q \rangle]) \\
&\lfloor !P \rfloor CTfrGHB\sigma = \lfloor P \rfloor CTfr'GHB(\sigma \cdot [r \mapsto \perp])
\end{aligned}
$$

Second, when the nonce or timestamp generation process is encountered, we add it into the execution trace M respectively. For the nonce generation process, we indicate its generation by adding a new event to the premises and insert it into U because nonces are random numbers. For the timestamp generation process, we add a timing constraint to describe the chronological order of timestamps as well as a $know$ event to show that the adversary can control the timing of process execution. When VR is applicable, $B_d(\mathbb{t}, \mathbb{t}', c)$ is $-p_c \leq \mathbb{t} - \mathbb{t}' \leq p_c$. When SR is applicable, $B_d(\mathbb{t}, \mathbb{t}', c)$ is $\mathbb{t} = \mathbb{t}' + d_c$.

$$
\begin{aligned}
&\lfloor \nu n.P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfr'G(H \uplus new([n], loc()) \uplus unique([n], loc(), f)) \\
&\qquad B(\sigma \cdot [r \mapsto \langle [n], r' \rangle]) \\
&\lfloor \mu t.P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor (C \circ \langle c_g, \mathbb{t} \rangle)(T \uplus \langle t, t \rangle) fr'G(H \uplus know(\mathbb{t}, \mathbb{t})) \\
&\qquad (B \cap C(c_g) \leq \mathbb{t})(\sigma \cdot [r \mapsto \langle \mathbb{t}, r' \rangle]) \\
&\lfloor \mu t : c.P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor (C \circ \langle c, \mathbb{t} \rangle \circ \langle c_g, \mathbb{t}' \rangle)(T \uplus \langle \mathbb{t}, \mathbb{t}' \rangle) r'G(H \uplus know(\mathbb{t}, \mathbb{t}')) \\
&\qquad (B \cap C(c) \leq \mathbb{t} \cap C(c_g) \leq \mathbb{t}' \cap B_d(\mathbb{t}, \mathbb{t}', c))(\sigma \cdot [r \mapsto \langle \langle t, t' \rangle, r' \rangle])
\end{aligned}
$$

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 specification becomes more flexible and accurate.

$$
\begin{aligned}
&\lfloor \text{if } m_1 = m_2 \text{ then } P \text{ else } Q \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrGHB(\sigma \cdot mgu(m_1, m_2)) \\
&\quad \cup \lfloor Q \rfloor CTfr(G \wedge m_1 \neq m_2)HB\sigma \\
&\lfloor \text{if } B_0 \text{ then } P \text{ else } Q \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrGH(B \cap B_0)\sigma \\
&\quad \cup (\cup_{c \in B_0} \lfloor Q \rfloor CTfrGH(B \cap \neg c)\sigma) \\
&\lfloor \text{wait } \mu t \text{ until } B_t \text{ then } P \rfloor CTmGHB\sigma \\
&\quad = \lfloor P \rfloor (C \circ \langle c_g, t \rangle)(T \uplus \langle t, t \rangle) r' G(H \uplus know(\mathbb{t}, \mathbb{t})) \\
&\qquad (B \cap B_t \cap C(g_c) \leq \mathbb{t})(\sigma \cdot [r \mapsto \langle \mathbb{t}, r' \rangle]) \\
&\lfloor \text{wait } \mu t : c \text{ until } B_t \text{ then } P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor (C \circ \langle c, \mathbb{t} \rangle \circ \langle c_g, \mathbb{t}' \rangle)(T \uplus \langle \mathbb{t}, \mathbb{t}' \rangle) r' G(H \uplus know(\mathbb{t}, \mathbb{t}')) \\
&\qquad (B \cap B_t \cap C(c) \leq \mathbb{t} \cap C(g_c) \leq \mathbb{t}' \cap B_d(\mathbb{t}, \mathbb{t}', c))(\sigma \cdot [r \mapsto \langle \langle \mathbb{t}, \mathbb{t}' \rangle, r' \rangle]) \\
&\text{Given function } f \text{ defined as } f(m'_1, \ldots, m'_n) \Rightarrow m'@D, \text{ we have} \\
&\lfloor \text{let } x = f(m_1, \ldots, m_n) \text{ then } P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrG(H \cup H')B(\sigma \cdot \{x \mapsto m'\} \cdot mgu(\langle m_1, \ldots, m_n \rangle, \langle m'_1, \ldots, m'_n \rangle))
\end{aligned}
$$

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 premise event as a requirement for the adversary to know that message. On the other hand, we generate a *time logic rule* for every network output, representing that the output will be known to the adversary when it is sent out.

$$
\begin{aligned}
&\lfloor in(x).P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor (C \circ \langle c_g, \mathbb{t} \rangle) Tr' G(H \uplus know(x, \mathbb{t}')) \\
&\qquad (B \cap C(c_g) \leq \mathbb{t} \cap \mathbb{t}' \leq \mathbb{t})(\sigma \cdot [r \mapsto \langle x, r' \rangle]) \\
&\lfloor \overline{out}(m).P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrGHB\sigma \\
&\quad \uplus ([\, G \,]\ H \multimap [\, B \cap \mathbb{t} - C(c_g) \geq \S p_n \,] \rightarrowtail know(m, \mathbb{t})) \cdot \sigma
\end{aligned}
$$

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

$$
\begin{aligned}
&\lfloor \text{check } m \text{ as unique then } P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrG(H \uplus unique(m, loc(), f))B\sigma
\end{aligned}
$$

Sixth, three types of authentication events can be engaged in the process. The $join$ event have the same arguments as of the $join$ expression in the calculus. However, for the $init$ and $accept$ events, although their meanings are preserved in the timed logic

rules, in order to check the injective authentication properties, we add an additional argument $[d]$ to represent the session id. The $init$ and $join$ events are added into the rule premises. The $accept$ events act as the rule conclusions.

$$
\begin{aligned}
&\lfloor init(m)@t.P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrG(H \uplus new([d], loc()) \\
&\qquad \uplus unique([d], loc(), f) \uplus init([d], m, T(\mathbb{t})))B\sigma \\
&\lfloor join(m)@t.P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrG(H \uplus new([d], loc()) \\
&\qquad \uplus unique([d], loc(), f) \uplus join([d], m, T(\mathbb{t})))B\sigma \\
&\lfloor accept(m)@t.P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrGHB\sigma \uplus ([\ G\ ]\ H \uplus new([d], loc()) \\
&\qquad \uplus unique([d], loc(), f) \multimap [\ B\ ] \rightarrowtail accept([d], m, T(\mathbb{t}))) \cdot \sigma
\end{aligned}
$$

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 [16], 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.

$$
\begin{aligned}
&\lfloor open(m).P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrG(H \uplus open(m))B\sigma \\
&\lfloor secrecy(m).P \rfloor CTfrGHB\sigma \\
&\quad = \lfloor P \rfloor CTfrGHB\sigma \\
&\qquad \uplus ([\ G\ ]\ H \uplus know(m, \mathbb{t}) \multimap [\ B\ ] \rightarrowtail leak(m)) \cdot \sigma
\end{aligned}
$$

Appendix B: TESLA Protocol Specification

TESLA [22,21] is short for Timed, Efficient, Streaming, Loss-tolerant Authentication protocol. It can provide efficient authenticated broadcast over lossy channels. Generally, it consists of many resource constrained receivers and a relatively powerful sender.

Protocol Description. The goal of TESLA is to transfer a set of messages $\{M_j \mid j \in [0 \dots n]\}$ from a sender *S* to a receiver *R* in an authenticated manner. Since *R* have limited computing power, *S* cannot adopt signature for authentication purpose because of the large computing overhead. As a result, *S* computes hash values for messages with hash keys and uses these keys for authentication. Specifically, *S* divides the message transmission time into several continuous intervals. Each interval has the same length of p_d ($p_d > 0$). Then, *S* sends the messages with their hash values in a time interval i and reveals the corresponding hash keys in a later time interval $i + d$. That is, *S* sends $\langle M_j, mac(M_j, k_i)\rangle$ in the i-th time interval and reveals the key k_i in the interval $i + d$. Since only *S* knows k_i before k_i is revealed, when k_i is check to be a hash key from *S*, $\langle M_j, mac(M_j, k_i)\rangle$ should be sent from *S*. In order to check the authenticity of the hash keys, TESLA requires these keys to form a chain such that k_i can be computed by k_{i+1} with a one-way function. Hence, when *S* can authenticate the first key k_0 to *R*, *R* can use k_0 to authenticate newly received hash keys. Additionally, using this method, even if some hash key k_i is lost, once k_{i+x} ($x > 0$) is received by *R*, k_i can be computed from k_{i+x} for authentication. In order to provide sound security, *S* in TESLA does not send the hash keys directly. Instead, it sends the hash key generators $\{k'_i\}$ and uses the generators to compute the actual hash keys $\{k_i\}$.

1. Sender Setup. In this step, *S* needs to decide the number of time intervals for its life-cycle, and then prepares the key chain for authentication. Firstly, *S* generates a random nonce k'_n as the hash key generator of the final interval n and then computes the generators $k'_i = mac(k'_{i+1}, 0)$ for other intervals $i \in [0 \dots n-1]$. The symmetric encryption key of interval i can be computed from k'_i as $k_i = mac(k'_i, 1)$.

2. Bootstrapping a new Receiver. Once *S* is properly set up, it needs to bootstrap new receivers with an initial authentication packet. This packet is a signature from *S* that specifies the beginning interval i and the corresponding key generator k'_{i-d}.

3. Clock Synchronization. Unlike WMF and many other protocols, TESLA does not assume perfect clock synchronization. It rather requires loose time synchronization between *S* and *R*, where *R* knows the upper-bound of the local clock drift δ between *S* and *R*. In order to obtain the upper-bound, TESLA adopts the following protocol with two messages.

(1) *R* reads its current time as t_r and generates nonce n
$R \rightarrow S : n$
(2) *S* reads its current time as t_s
$S \rightarrow R : \{t_s, n\}_{sk_s}$

Firstly, *R* reads its current time as t_r, generates a nonce (a random number) n and sends n to *S*. Secondly, *S* reads its current time as t_s, sign t_s and n with its private signing key sk_s and sends the signature back to *R*. When *R* receives the signature from *S*, *R* can

$Sender \triangleq$

01 $!(in(n).\mu t_s : c_s.\overline{out}(sign(\langle syn, t_s, n\rangle, sk_s)).0)$ *(Clock Sync)*

02 $|$ $(\nu k'_2$.let $k'_1 = mac(k'_2, 0)$ then let $k'_0 = mac(k'_1, 0)$ then *(Sender Setup)*

03 let $k_1 = mac(k'_1, 1)$ then let $k_2 = mac(k'_2, 1)$ then

04 $\mu t_0 : c_s.($

05 $!(\overline{out}(sign(\langle boot, t_0, k'_0\rangle, sk_s)).0)$ *(Bootstrapping)*

06 $|$ $!(\nu m.\mu t'_s : c_s.$ *(Send Packets)*

07 if $t'_s < t_0 + p_d$ then

08 $init(pk_s, m, k_1)@t'_s.\overline{out}(\langle m, mac(k_1, m)\rangle).0$

09 else if $t'_s < t_0 + 2 \times p_d$ then

10 $init(pk_s, m, k_2)@t'_s.\overline{out}(\langle m, mac(k_2, m)\rangle).\overline{out}(k'_1).0)$

11 $|$!(wait $\mu t_s : c_s$ until $t_s \geq t_0 + 2 \times p_d$ then $\overline{out}(k'_2).0)))$

$Receiver \triangleq$

12 $\mu t_r : c_r.\nu n.\overline{out}(n).$ *(Clock Sync)*

13 $in(sync)$.let $\langle =syn, t_s, =n\rangle = check(sync, pk_s)$ then

14 $in(sig)$.let $\langle =boot, t_0, k'_0\rangle = check(sig, pk_s)$ then *(Bootstrapping)*

15 $!(in(payload)$.let $\langle m, ck\rangle = payload$ then *(Receiver Tasks)*

16 $\mu t'_r : c_r$.let $t_s^{bound} = t'_r + t_s - t_r$ then

17 if $t_s^{bound} < t_0 + p_d$ then $in(k')$.if $k'_0 = mac(k', 0)$ then

18 if $ck = mac(mac(k', 1), m)$ then $accept(pk_s, m, mac(k', 1))@t'_r.0$

19 else if $t_s^{bound} < t_0 + 2 \times p_d$ then $in(k')$.if $k'_0 = mac(mac(k', 0), 0)$ then

20 if $ck = mac(mac(k', 1), m)$ then $accept(pk_s, m, mac(k', 1))@t'_r.0)$

Fig. 1. TESLA Model in Timed Applied π-calculus

be sure that δ has an upper-bound of $t_s - t_r$. Thereafter, when *R* receives a message from *S* at its local time t'_r, he can claim that the current time of *S* is upper-bounded by $t'_r + t_s - t_r$.

4. Sending Authenticated Packets. In this step, *S* can send messages to *R* with authenticated packets. The packet P_j for a message M_j sent in interval i can be constructed as $\langle M_j, mac(k_i, M_j), k'_{i-d}\rangle$. Once *R* receives the packet $P_j = \langle M_j, mac(k_i, M_j), k'_{i-d}\rangle$ at time t'_r, it computes an upper-bound of *S*'s local clock t'_s as $t'_r + t_s - t_r$. *R* then ensures that *S* has not revealed the k_i yet at its local time t'_s (based on the results from *Clock Synchronization*). However, *R* cannot verify the authenticity of P_j until k'_i is revealed in a later packet. Hence, *R* stores P_j in its memory and waits for a key generator k'_x ($x \geq i$) to generate k_i. When the last verified key generator is k'_v, *R* can verify a newly received key generator k'_x by checking $k'_v = mac^{x-v}(k'_x, 0)$.

TESLA Model. In this section, we model TESLA illustrated above using *timed applied π-calculus*. Firstly, we declare two local clocks for *S* and *R* as c_s and c_r respectively. By assumption, c_s's clock drift bounded from above by a parameter p_s, and c_r's clock drift is bounded by a parameter p_r. In our model, sk_s is the private signing key of *S* and $pk_s = pk(sk_s)$ is the corresponding public key. Additionally, we assume that $d = 1$ in TESLA, i.e., *S* reveals the key generator k'_i at interval $i + 1$ for every message sent at interval i. For demonstration purpose, the hash key chain in our model consists of three hash keys. The analysis of TESLA with longer key chains are available in Section 5.

Based on above settings and assumptions, we could model the process of *S* as $Sender$ and the process of *R* as $Receiver$ as shown in Figure 1. Then, the overall protocol then can be modeled as $TESLA \triangleq Sender|!Receiver|!\overline{out}(pk_s)$.

The process $Sender$ consists of two sub-processes. Line 1 corresponds to the sub-process for clock synchronization and the process from Line 2 to Line 11 is the main process of TESLA. From Line 2 to Line 3, *S* generates a nonce k_2' as the hash key generator for the last interval and then computes k_0', k_1', k_1 and k_2 for later usage. k_0' is the initial key generator that will be authenticated to the receivers. Then, *S* records the starting time of TESLA as t_0 at Line 4 and broadcasts the signature of $\langle t_0, k_0' \rangle$ at Line 5. Based on the current local time, *S* chooses different hash keys to authenticate the messages from Line 6 to Line 10. *S* reveals the last hash key k_2 at Line 11 when all time intervals are over.

The process $Receiver$ does the corresponding tasks to the process $Sender$. Firstly, it synchronizes its clock from Line 1 to Line 2. After the synchronization, whenever it reads a timestamp t_r' from its local clock, it can be sure that the local time of *S* is upper-bounded by $t_r' + t_s - t_r$. Then, at Line 14, it receives the signature of the beginning time t_0 and the initial key generator k_0' from *S*. From Line 15 to Line 20, *R* receives the messages from *S* in an authenticated way. When it gets a message m and its hash value ck from the network, it ensures that ck has not been revealed by *S*. Later, when *R* obtains the key generator k', it checks k' using k_0' and then checks ck using k'. When k' and ck are correct, *R* claims that the message m is legitimately sent by *S*.

Timed Properties. TESLA ensures that every message accepted by *R* is sent by *S* previously. Hence, we have the following non-injective authentication property.

$$accept(pk_s, m, k)@t_r \leftarrow\!\!\{\ t_s \leq t_r\ \}\!\!- init(pk_s, m, k)@t_s$$

Appendix C: Frequently Used Cryptographic Functions

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)
	$mac(k, m)$	(compute mac value)
Tuple	$tuple_n(m_1, \ldots, m_n)$	(construct tuple)
	$\forall i \in \{1 \ldots n\}$: $get_n^i(tuple_n(m_1, \ldots, m_n)) \Rightarrow m_i$	(extract tuple)

Table 4. Cryptographic Function Definitions

Appendix D: Verification of CWMF

In this section, we focus on finding the attack of CWMF when we assume VR. We use the following *timed logic rules* to express the *timed applied π-calculus* process of CWMF. First, P_r is written as Rule (3).

$$[\ u \neq A[] \wedge u \neq B[]\ ]\ know(u, \mathbb{t}_1) \dashv[\ \mathbb{t} - \mathbb{t}_1 \geq \S p_n\ ]\mapsto know(key(u), \mathbb{t}) \tag{3}$$

Secondly, P_a is written as Rule (4), where $p_a > 0$ is the maximum drift of c_a.

$$\begin{aligned} &new([k], l_a[]), unique([k], l_a[], \langle r, [k], \langle \mathbb{t}_a, \mathbb{t}'_a\rangle\rangle) \\ &\quad, know(r, \mathbb{t}_1), know(\mathbb{t}_a, \mathbb{t}_a), init([k], (A[], r, [k]), \mathbb{t}'_a) \\ &\quad \dashv[\ \mathbb{t}_1 \leq \mathbb{t}'_a \wedge \mathbb{t} - \mathbb{t}'_a \geq \S p_n \wedge |\mathbb{t}_a - \mathbb{t}'_a| \leq \S p_a\ ]\mapsto \\ &\qquad know(\langle A[], enc_s(\langle \mathbb{t}_a, r, [k], tag_1[]\rangle, key(A[]))\rangle, \mathbb{t}) \end{aligned} \tag{4}$$

Thirdly, P_s is written as Rule (5), where $p_s > 0$ is the maximum drift of c_s.

$$\begin{aligned} &know(\langle i, enc_s(\langle \mathbb{t}_a, r, k, tag_1[]\rangle, key(i))\rangle, \mathbb{t}_1), new([n_s], l_a[]) \\ &\quad, unique([n_s], l_a[], \langle\langle i, enc_s(\langle \mathbb{t}_a, r, k, tag_1[]\rangle, key(i))\rangle, k, \langle \mathbb{t}_s, \mathbb{t}'_s\rangle\rangle) \\ &\quad, know(\mathbb{t}_s, \mathbb{t}_s), join([n_s], (i, r, k), \mathbb{t}'_s) \\ &\quad \dashv[\ \mathbb{t}_1 \leq \mathbb{t}'_s \wedge \mathbb{t} - \mathbb{t}'_s \geq \S p_n \wedge \mathbb{t}_s - \mathbb{t}_a \leq \S p_m \wedge |\mathbb{t}_s - \mathbb{t}'_s| \leq \S p_s\ ]\mapsto \\ &\qquad know(enc_s(\langle \mathbb{t}_s, i, k, tag_2[]\rangle, key(r)), \mathbb{t}) \end{aligned} \tag{5}$$

Fourthly, P_b is written as Rule (6), where $p_b > 0$ is the maximum drift of c_b.

$$\begin{aligned} &know(enc_s(\langle \mathbb{t}_s, i, k, tag_2[]\rangle, key(B[])), \mathbb{t}_1), know(\mathbb{t}_b, \mathbb{t}_b), new([n_b], l_a[]) \\ &\quad, unique([n_b], l_a[], \langle enc_s(\langle \mathbb{t}_s, i, k, tag_2[]\rangle, key(B[])), [n_b], \langle \mathbb{t}_b, \mathbb{t}'_b\rangle\rangle) \\ &\quad, unique(k, db[], \langle enc_s(\langle \mathbb{t}_s, i, k, tag_2[]\rangle, key(B[])), [n_b], \langle \mathbb{t}_b, \mathbb{t}'_b\rangle\rangle) \\ &\quad \dashv[\ \mathbb{t}_1 \leq \mathbb{t}'_b \wedge \mathbb{t}_b - \mathbb{t}_s \leq \S p_m \wedge |\mathbb{t}_b - \mathbb{t}'_b| \leq \S p_b\ ]\mapsto \\ &\qquad accept([n_b], \langle i, B[], k\rangle, \mathbb{t}'_s) \end{aligned} \tag{6}$$

Fifthly, P_p can be expressed as Rule (7) and Rule (8).

$$\dashv[\ \ ]\mapsto know(A[], \mathbb{t}) \tag{7}$$

$$\dashv[\ \ ]\mapsto know(B[], \mathbb{t}) \tag{8}$$

Lastly, symmetric encryption and symmetric decryption can be written as Rule (9) and Rule (10) respectively.

$$know(m, \mathbb{t}_1), know(k, \mathbb{t}_2) \dashv[\ \mathbb{t}_1 \leq \mathbb{t} \wedge \mathbb{t}_2 \leq \mathbb{t}\ ]\mapsto know(enc_s(m, k), \mathbb{t}) \tag{9}$$

$$know(enc_s(m, k), \mathbb{t}_1), know(k, \mathbb{t}_2) \dashv[\ \mathbb{t}_1 \leq \mathbb{t} \wedge \mathbb{t}_2 \leq \mathbb{t}\ ]\mapsto know(m, \mathbb{t}) \tag{10}$$

When we compose Rule(4) to Rule (5), we can obtain the following rule.

$$
\begin{aligned}
&new([k], l_a[]), unique([k], l_a[], \langle r, [k], \langle \mathbb{t}_a, \mathbb{t}'_a \rangle \rangle) \\
&\quad , know(r, \mathbb{t}_1), know(\mathbb{t}_a, \mathbb{t}_a), init([k], (A[], r, [k]), \mathbb{t}'_a), new([n_s], l_a[]) \\
&\quad , unique([n_s], l_a[], \langle \langle A[], enc_s(\langle \mathbb{t}_a, r, [k], tag_1[] \rangle, key(A[])) \rangle, [k], \langle \mathbb{t}_s, \mathbb{t}'_s \rangle \rangle) \\
&\quad , know(\mathbb{t}_s, \mathbb{t}_s), join([n_s], (A[], r, [k]), \mathbb{t}'_s) \\
&\quad \dashv[\, \mathbb{t}_1 \leq \mathbb{t}'_a \wedge \mathbb{t}'_s - \mathbb{t}'_a \geq \S p_n \wedge |\mathbb{t}_a - \mathbb{t}'_a| \leq \S p_a \\
&\quad \wedge \mathbb{t} - \mathbb{t}'_s \geq \S p_n \wedge \mathbb{t}_s - \mathbb{t}_a \leq \S p_m \wedge |\mathbb{t}_s - \mathbb{t}'_s| \leq \S p_s \,] \mapsto \\
&\qquad know(enc_s(\langle \mathbb{t}_s, A[], [k], tag_2[] \rangle, key(r)), \mathbb{t})
\end{aligned} \tag{11}
$$

Then, we can compose Rule (11) to Rule (6) and obtain the following rule.

$$
\begin{aligned}
&new([k], l_a[]), unique([k], l_a[], \langle B[], [k], \langle \mathbb{t}_a, \mathbb{t}'_a \rangle \rangle) \\
&\quad , know(B[], \mathbb{t}_1), know(\mathbb{t}_a, \mathbb{t}_a), init([k], (A[], B[], [k]), \mathbb{t}'_a), new([n_s], l_a[]) \\
&\quad , unique([n_s], l_a[], \langle \langle A[], enc_s(\langle \mathbb{t}_a, B[], [k], tag_1[] \rangle, key(A[])) \rangle, [k], \langle \mathbb{t}_s, \mathbb{t}'_s \rangle \rangle) \\
&\quad , know(\mathbb{t}_s, \mathbb{t}_s), join([n_s], (A[], B[], [k]), \mathbb{t}'_s) \\
&\quad , know(\mathbb{t}_b, \mathbb{t}_b), new([n_b], l_a[]) \\
&\quad , unique([n_b], l_a[], \langle enc_s(\langle \mathbb{t}_s, A[], [k], tag_2[] \rangle, key(B[])), [n_b], \langle \mathbb{t}_b, \mathbb{t}'_b \rangle \rangle) \\
&\quad , unique([k], db[], \langle enc_s(\langle \mathbb{t}_s, A[], [k], tag_2[] \rangle, key(B[])), [n_b], \langle \mathbb{t}_b, \mathbb{t}'_b \rangle \rangle) \\
&\quad \dashv[\, \mathbb{t}_1 \leq \mathbb{t}'_a \wedge \mathbb{t}'_s - \mathbb{t}'_a \geq \S p_n \wedge |\mathbb{t}_a - \mathbb{t}'_a| \leq \S p_a \\
&\quad \wedge \mathbb{t}'_b - \mathbb{t}'_s \geq \S p_n \wedge \mathbb{t}_s - \mathbb{t}_a \leq \S p_m \wedge |\mathbb{t}_s - \mathbb{t}'_s| \leq \S p_s \\
&\quad \wedge \mathbb{t}_b - \mathbb{t}_s \leq \S p_m \wedge |\mathbb{t}_b - \mathbb{t}'_b| \leq \S p_b \,] \mapsto \\
&\qquad accept([n_b], \langle A[], B[], [k] \rangle, \mathbb{t}'_s)
\end{aligned} \tag{12}
$$

Then, we can use Rule (8) to fulfill a premise of Rule (12).

$$
\begin{aligned}
&new([k], l_a[]), unique([k], l_a[], \langle B[], [k], \langle \mathbb{t}_a, \mathbb{t}'_a \rangle \rangle) \\
&\quad know(\mathbb{t}_a, \mathbb{t}_a), init([k], (A[], B[], [k]), \mathbb{t}'_a), new([n_s], l_a[]) \\
&\quad , unique([n_s], l_a[], \langle \langle A[], enc_s(\langle \mathbb{t}_a, B[], [k], tag_1[] \rangle, key(A[])) \rangle, [k], \langle \mathbb{t}_s, \mathbb{t}'_s \rangle \rangle) \\
&\quad , know(\mathbb{t}_s, \mathbb{t}_s), join([n_s], (A[], B[], [k]), \mathbb{t}'_s) \\
&\quad , know(\mathbb{t}_b, \mathbb{t}_b), new([n_b], l_a[]) \\
&\quad , unique([n_b], l_a[], \langle enc_s(\langle \mathbb{t}_s, A[], [k], tag_2[] \rangle, key(B[])), [n_b], \langle \mathbb{t}_b, \mathbb{t}'_b \rangle \rangle) \\
&\quad , unique([k], db[], \langle enc_s(\langle \mathbb{t}_s, A[], [k], tag_2[] \rangle, key(B[])), [n_b], \langle \mathbb{t}_b, \mathbb{t}'_b \rangle \rangle) \\
&\quad \dashv[\, \mathbb{t}'_s - \mathbb{t}'_a \geq \S p_n \wedge |\mathbb{t}_a - \mathbb{t}'_a| \leq \S p_a \\
&\quad \wedge \mathbb{t}'_b - \mathbb{t}'_s \geq \S p_n \wedge \mathbb{t}_s - \mathbb{t}_a \leq \S p_m \wedge |\mathbb{t}_s - \mathbb{t}'_s| \leq \S p_s \\
&\quad \wedge \mathbb{t}_b - \mathbb{t}_s \leq \S p_m \wedge |\mathbb{t}_b - \mathbb{t}'_b| \leq \S p_b \,] \mapsto \\
&\qquad accept([n_b], \langle A[], B[], [k] \rangle, \mathbb{t}'_s)
\end{aligned} \tag{13}
$$

According to the authentication property, we need to ensure $\mathbb{t}'_b - \mathbb{t}'_a \leq \S p_m \wedge \mathbb{t}'_s - \mathbb{t}'_b \leq \S p_m$ in Rule (13). Hence, $\S p_a + \S p_m + \S p_s \leq \S p_m \wedge \S p_b + \S p_m + \S p_s \leq \S p_m$ is required for protocol security. Then, we have $\S p_a + \S p_s \leq 0 \wedge \S p_b + \S p_s \leq 0$. However,

we assume that the clock drift exists, i.e., $\S p_a$, $\S p_s$ and $\S p_b$ are positives. As a result, we cannot find any value for them to ensure the authentication property, which can be concluded as an attack for CWMF under VR.