First Results of a Formal Analysis of the Network Time Security Specification

Kristof Teichel[1(✉)], Dieter Sibold[1], and Stefan Milius[2]

[1] Physikalisch-Technische Bundesanstalt, Bundesallee 100, 38116 Braunschweig, Germany
{kristof.teichel,dieter.sibold}@ptb.de
[2] Chair for Theoretical Computer Science, Friedrich-Alexander Universität Erlangen-Nürnberg, Martensstr. 3, 91058 Erlangen, Germany
stefan.milius@fau.de

Abstract. This paper presents a first formal analysis of parts of a draft version of the Network Time Security specification. It presents the protocol model on which we based our analysis, discusses the decision for using the model checker ProVerif and describes how it is applied to analyze the protocol model. The analysis uncovers two possible attacks on the protocol. We present those attacks and show measures that can be taken in order to mitigate them and that have meanwhile been incorporated in the current draft specification.

Keywords: Time synchronization · Security protocols · Formal verification · Model checking · ProVerif

1 Introduction

In networked infrastructures, time synchronization protocols are often used to synchronize clocks. Many technical infrastructures require reliable time synchronization. Therefore, it is essential to be able to secure time synchronization from malicious attacks. One approach in this area is the Network Time Security (NTS) specification [29], which aims to provide sufficiently generic mechanisms to secure two of the most important time synchronization protocols: the Network Time Protocol (NTP) [21] and the Precision Time Protocol (PTP) [14]. Since the NTS specification is still under development, it is beneficial to apply some form of formal protocol analysis. This might serve to find weaknesses which still exist, as well as to establish which of its components can already be considered secure.

It has been asserted that the difficulty of the formal analysis of a security protocol increases with its dependence on timing [3,4,12]. This is even more the case for NTS, which not only depends on timing, but it additionally distinguishes itself from other timing-dependent security protocols because the underlying secured time synchronization protocol actively influences the clocks of its participants. A thorough analysis has to take into account the impact of timing on the security properties of the specification.

L. Chen and S. Matsuo (Eds.): SSR 2015, LNCS 9497, pp. 218–245, 2015.
DOI: 10.1007/978-3-319-27152-1_12

In this paper, we limit the scope and present the first steps in the analysis of NTS, namely an evaluation of parts of the specification with timing aspects abstracted away. For the analysis, we apply the model checker ProVerif [6] to analyze version 03 of the NTS specification. This unveils two possible attacks. The corresponding vulnerabilities have been removed in the next version of the NTS specification.

The paper is organized as follows. Section 2 provides an overview of the role of timing in the context of secure time synchronization. Furthermore it discusses criteria for different stages of analysis of secure time synchronization specifications, explaining our chosen scope, and it illuminates the choice of the model checker ProVerif as the formal analysis tool for the particular context of this paper. In Sect. 3, we introduce the notation and modeling assumptions that are required for the rest of the paper. Section 4 provides some background on the NTS specification, and presents the protocol steps that are modeled for our analysis. Section 5 describes the resulting model for ProVerif. The obtained results (attacks and countermeasures) are discussed in Sect. 6, and Sect. 7 concludes the paper.

2 Security for Packet-Based Time Synchronization

2.1 Time Synchronization Methods

Figure 1 displays a two-way time transfer message exchange that a client (Alice) and a server (Bob) can use to achieve clock synchronization. At time t_1, Alice initiates a time request message (a) to Bob, where it arrives, after a delay of $\xi\delta$, at time t_2. At time t_3, Bob sends his time response (b) back to Alice, where it arrives after a delay of $(1-\xi)\delta$, at time t_4. The timestamps $T_1, \ldots, T_4$ are added to the exchanged messages by Alice and Bob respectively.[1] Upon arrival of the time response, Alice can calculate her time offset Δ with respect to Bob and the network round-trip delay δ to

$$\Delta = \frac{(T_1 + T_4) - (T_2 + T_3)}{2} + \left(\xi - \frac{1}{2}\right)\delta, \quad 0 < \xi < 1\,, \tag{1}$$

$$\delta = (T_4 - T_1) - (T_3 - T_2)\,, \tag{2}$$

where δ denotes the round-trip travel time and the asymmetry parameter ξ quantifies the degree of asymmetry in the travel time between the time request and time response messages [16]. Without specific knowledge about the network infrastructure components on the path between Alice and Bob the asymmetry parameter is *a priori* unknown. Hence, in most cases symmetric message delays are assumed, i. e. $\xi = 0.5$. In this case, the second term in the right-hand side of (1) vanishes and Alice can calculate the time offset Δ. Since the ratio between the message delays is important for the time offset calculation, the cryptographic means employed by any security specification must not unduly influence the asymmetry parameter ξ [22].

[1] Timestamp T_i corresponds to the reading of the respective system clock at time t_i.

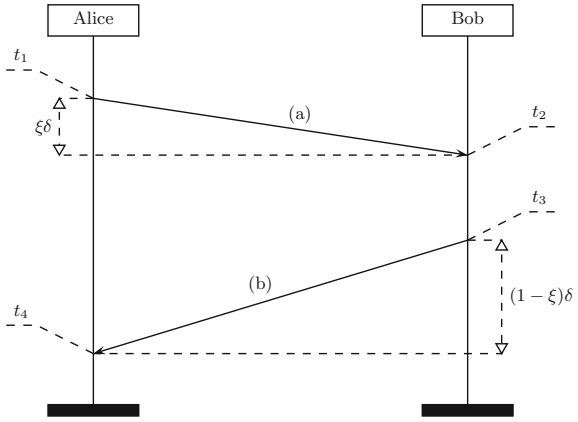

Fig. 1. This is a schematic depiction of a two-way time transfer message exchange that is used for unicast time synchronization between two participants. In the depiction, δ denotes the round-trip time of the complete message exchange and ξ, with $0 < \xi < 1$, quantifies its asymmetry.

There is also a necessity of considering attacks on the timeliness of the exchanged messages, most importantly delay attacks (also called "pulse delay attacks") [11,22]. This kind of attack is characterized by the fact that an adversary can delay the delivery of a message, by first preventing it from being delivered and later replaying it. Such an attack degrades the performance of the time synchronization, it is very simple to perform, and it is not preventable by purely cryptographic means since the content of the exchanged messages does not have to be modified by the adversary [22].

2.2 Criteria for Different Stages of Analysis

We first discuss the criteria underlying the choice of methods and tools for our analysis. It is highly desirable that a full evaluation of a secure time synchronization protocol such as NTS-secured NTP take into account clocks and time as well as properties of cryptographic security protocols. Not only does the information transmitted by the time synchronization protocol concern time data, but the protocol's purpose is to achieve time synchronization. Consequently, the amount of time that it takes for messages to be transmitted is important. Furthermore, time and clock information is relevant for the TESLA protocol [25], which is employed in the NTS specification for broadcast/multicast messages to ensure integrity protection. As a consequence, there is complex interaction between the synchronization operations that the time synchronization protocol performs and the security measures that it uses. All this indicates that the methods and tools used for the analysis of a specification like NTS should ideally be able to consider the notion of time and clocks. Also, the model for time and clocks should be as detailed as possible, without compromising the automated analysis of the model.

However, for parts of the NTS specification, a clean distinction between timing and security-related properties is possible. Specifically, the majority of goals for the unicast mode of NTS, including server authentication, secure key exchange, and integrity protection of time synchronization messages, can be evaluated without considering time or clocks as a part of the model. An analysis using this self-contained scope can already be regarded as a substantial step in the analysis of the whole protocol.

The analysis in this paper was performed at an early stage of the specification process of NTS [29, version 03]. At such a stage, an important criterion is that the methods and tools used for the analysis can find existing weaknesses quickly and point them out clearly. In this case, the specification efforts can benefit from the results of a formal analysis, if the analysis unveils security vulnerabilities in the specification which subsequently can be communicated and considered for the next redrafting of the specification.

These considerations lead us to present an analysis of the unicast mode of NTS, with all of its timing and clock aspects abstracted away, matching the above-mentioned self-contained scope. Since this scope does not require consideration of timing or clocks, the ability to do so can be neglected as a requirement in the choice of methods and tools. We are aware that with this scope, the detection of delay attacks is not possible. The more intricate interactions between cryptographic security and timing would have to be included for a full evaluation of NTS. For such a full evaluation, one could also include an analysis of the broadcast mode of NTS. Further research in that direction is in progress and will be published at a later time [19].

2.3 Choice of Tool for the Analysis

Several approaches are available to perform a formal analysis of a security protocol, the most established being theorem proving and model checking [1]. We first applied Paulson's Inductive Approach [24] for our analysis of NTS. This approach is based on theorem proving and works with the theorem proving tool Isabelle [23]. Using it was motivated, inter alia, by developments which permit physical properties to be considered, especially timing and clocks [4]. Being based on theorem proving, the approach has the drawback that failure to construct a security proof is inconclusive. The reason might either be that the proof is too difficult to find, although the investigated specification is secure, or it might be that the specification is simply not secure. Although counterexample finders exist for Isabelle [7], we were unable to obtain any attack traces for our analysis with the help of such tools. This potential ambiguity is not suitable if the analysis shall support the ongoing drafting process of the considered security specification. Moreover, we found that the pace at which this approach delivers results is slow, at least for non-experts in theorem proving.

The model checking approach offers faster and easier ways to identify vulnerabilities in protocol specifications. With regard to the scope discussed in Subsect. 2.2, we base our formal analysis in this paper on model checking by applying the designated protocol verifer ProVerif [6], which applies a variant of

the applied π-calculus with support of types [2,27] as its input language. Note that ProVerif does not support consideration of time and clocks for modeling: although there are time and clock related extensions of the π-calculus [28], none of them are applicable within ProVerif. Note also that it is impossible to simply reproduce the timing-related extension of the Inductive Approach for use with ProVerif. Its lack of support for time and clock consideration would constitute a serious disadvantage of using ProVerif for a full evaluation of NTS. However, for the scope as described in Sect. 2.2 this is negligible, especially since ProVerif fulfills the other criterion very well: it is tailored to be a verification tool for security protocols and offers help in the detection of attack scenarios. Since our results could readily be obtained with ProVerif, we did not investigate the use of other protocol verifiers such as Scyther [8] and Tamarin [18].

3 Basic Assumptions and Protocol Notation

In this section we establish assumptions to simplify the considered model and introduce the essential notation.

Assumption 1. There is exactly one trusted entity, called Trent. Every agent knows Trent's identity, as well as his public key K_T. Every honest agent trusts Trent completely. Certificates (see the list of cryptographic operators below) signed by Trent are assumed to authenticate agents beyond any doubt. In short form, Trent is denoted as T.

Remark 2. For the model, it is a helpful simplification to leave out any intermediate certificates. Therefore, certificate chains that certify an agent X and have Trent as their root authority are seen as equivalent to a certificate for X signed directly by Trent.

Assumption 3. There is exactly one attacker, called Mallory.[2] She complies with the Dolev-Yao attacker model [9], which means that she "controls the network." Explicitly, this means that the attacker has the following capabilities:

- She overhears and intercepts any message sent on the network. In particular, this also means that she can choose to prevent any message from being delivered in its original form.
- She can send messages to any agent on the network, claiming to possess any identity she chooses.
- She can synthesize messages by:
  - inventing new values (it is, however, out of her power to guess secret values like keys or nonces),
  - assembling multiple values known to her into a tuple value,

[2] The assumption that there is only one attacker is made for simplification. The assumed situation is equivalent to a situation where several attackers are cooperating, or to a situation where one attacker is being helped by one or more dishonest agents, see Reference [32].

- disassembling any tuple value that she knows into its single component values,
- applying any operator to any value known to her, possibly using any keys, as long as they are also in her knowledge.

In short form, Mallory is denoted as M.

Remark 4. Note that the Dolev-Yao model assumes cryptographic operations to be unbreakable. Thus, although the attacker can claim any chosen identity on a network level, it is still possible to verify authorship of a message by cryptographic means, through appropriate use of secrets.

Notation 5. For convenience and readability, we often use the following "box notation". Concatenation in box notation is displayed by writing the concatenated messages below each other in one box, separated by a dashed line; for example, the concatenation of values x_1 and x_2 is expressed as $\boxed{\begin{matrix} x_1 \\ \hdashline x_2 \end{matrix}}$. The box notation displays use of cryptographic operators as follows. The expression $\mathrm{Op}\left\langle \boxed{x} \right.$ evaluates to $\mathrm{Op}(x)$, where Op can be any one of the cryptographic operators presented below, and where x is usually a concatenated term.

Notation 6. We now provide some notation on cryptographic operators as used in the further presentation of the protocol. Note that a term inside square brackets [ ] generally denotes a key.

- The expression $\mathrm{Enc}[K](m)$ stands for the ciphertext that results from using asymmetric cryptography to encrypt the message m with the key K.
- Following the notion of asymmetric cryptography, it is assumed that for every key K, there exists an inverse key K^{-1}, such that both of the equations $\mathrm{Enc}[K^{-1}]\Big(\mathrm{Enc}[K](m)\Big) = m$ and $\mathrm{Enc}[K]\Big(\mathrm{Enc}[K^{-1}](m)\Big) = m$ are satisfied.
- The expression $\mathrm{Sign}[K](m)$ evaluates to $\mathrm{Sign}[K](m) = \mathrm{Enc}[K^{-1}](m)$, which describes the signature of the message m created in such a way that it can be validated with the key K.
- The expression Cert_X evaluates to $\mathrm{Cert}_X = \mathrm{Sign}[K_T]\left\langle \boxed{\begin{matrix} X \\ \hdashline K_X \end{matrix}} \right.$, which represents the certificate for X's public key K_X, issued by Trent.
- It is assumed that all participants have agreed to use a fixed cryptographic hash function h.[3] For a given key value K and message m, the expression $\mathrm{HMAC}[K](m)$ stands for the keyed-hash message authentication code computed over m, with K as key and h as the cryptographic hash function, as defined in RFC 2104 [15].

[3] Note that the NTS specification includes negotiation of hash functions as part of the protocol. However, as stated under Assumption 3, all cryptography is assumed to be unbreakable. Therefore, algorithm negotiation has been ignored for our analysis. It might be interesting to include these steps in future analysis, to look for downgrade attacks.

– The expression Seed_X represents an agent X's seed, a random secret value that is known only to X.
– The expression $\text{Cook}_X(v)$ evaluates to $\text{Cook}_X(v) = \text{HMAC}[\text{Seed}_X](v)$, which represents the cookie that agent X generates, using its own secret seed as well as the given input value v.

4 The Protocol Steps Under Analysis

4.1 The Network Time Security Project

The Network Time Security (NTS) specification aims at secure time synchronization over networks like the Internet. It is motivated by the fact that neither of the predominant time synchronization protocols, in particular the Network Time Protocol (NTP) [21] and the Precision Time Protocol (PTP) [14] currently provide adequate security mechanisms (see, e.g., Reference [26] for an analysis of the NTP's most current security measure, the Autokey protocol [20]). It has the goal of being usable to secure at least those two protocols [29]. Currently the specification is in the standardization process in the Internet Engineering Task Force (IETF).

Time synchronization in NTP unicast mode is secured via a secret value (called "cookie") which is unique for each client-server association. The nature of this cookie is such that the server can always deterministically re-generate it from an input value given by the client, while the client simply memorizes it. It is used as a key for generating a keyed-hash message authentication code (MAC) for each time synchronization response message going from the server to the client. Under the requirement that the cookie has been exchanged securely (its secrecy and authenticity need to be guaranteed), this procedure shall give authenticity and integrity guarantees for the time synchronization response messages. Having cookie and MAC generation based on keyed-hash mechanisms shall keep these steps fast enough to not significantly influence the quality of time synchronization [29].

4.2 Overview of the Protocol Sequence

We now describe the protocol messages and the protocol sequence. We present the protocol steps as specified in the NTS draft version 03, which constitutes the basis for the verification described in this paper. We only present those messages that are relevant for the scope of our analysis here. Furthermore, as discussed in Sect. 2.2, we abstract away any timing and clock aspects of the message exchanges that are presented.

The **certification** message exchange serves to supply the client Alice with the server Bob's certificate. The message exchange proceeds according to the protocol steps depicted in Table 1.

After the message exchange, Alice performs two checks. First, she verifies the validity of Cert_B. Second, she verifies the validity of the included signature. If all of this is successful, Alice trusts any subsequent message if it contains a signature that she is able to verify as originating from Bob.

Table 1. (Certification) These are the two message formats used for transmitting the certificate for server Bob (B) to client Alice (A) according to protocol version 0.3.0.

Name	Direction	Contents
client_cert	$A \rightarrow B:$	A
server_cert	$B \rightarrow A:$	$\boxed{\begin{array}{l}\text{Sign}[K_B](A) \\ \hdashline \text{Cert}_B\end{array}}$

The **cookie establishment** message exchange establishes the shared secret called the *cookie*, which is calculated by the server (deterministically and based on data supplied by the client) and then securely transmitted back to the client. The message exchange is performed according to the steps visible in Table 2.

After the message exchange, Alice decrypts the received response, and then performs two checks on the encrypted data. First, she validates that the nonce included in the response is the same one that she included in her request. Second, she confirms the validity of the signature included in the response. If all of this is successful, Alice trusts any subsequent message if it contains a MAC calculated with the cookie she received via this message exchange.

The purpose of the **unicast time synchronization** message exchange is to perform the actual time synchronization. Security is based on the secret cookie, which must be exchanged prior to this message exchange (via a cookie establishment exchange). The time synchronization message exchange follows the steps listed in Table 3, where S_A and S_B denote the time synchronization data transmitted by the client and the server, respectively. Note that with respect to the unicast time synchronization procedure described in Sect. 2 (see Fig. 1), S_A includes a timestamp of t_1, whereas S_B includes timestamps of t_1, t_2 and t_3.

After the message exchange, Alice performs two checks on the response. First, she validates that the nonce included in the response is the same one that she included in her request. Second, she confirms the validity of the MAC included in the response. If all of this is sucessful, Alice uses the received time data S_B for time synchronization.

5 Performing the Analysis

In order to use ProVerif for analysis of the NTS protocol, we model its participant roles (client and server) as processes in ProVerif's input language. The resulting specification is fed into ProVerif, which is able to analyze the protocol and prove reachability properties, correspondence assertions, as well as observational equivalence [6]. Note that having ProVerif check any single goal and, if applicable, generate an attack trace took merely a few seconds on a standard laptop computer.[4]

[4] The computer was running a 64 bit version of Windows 7 on an Intel i5 dual core at 2.6 GHz, with 8 GB of RAM.

Table 2. (Cookie Exchange) These are the two message formats used for the cookie exchange between client Alice (A) and server Bob (B) according to protocol version 0.3.0.

Name	Direction	Contents
client_cook	$A \rightarrow B$:	N / K_A
server_cook	$B \rightarrow A$:	$\mathrm{Enc}[K_A]\langle N \,/\, \mathrm{Cook}_B(h(K_A)) \,/\, \mathrm{Sign}[K_B]\langle N \,/\, \mathrm{Cook}_B(h(K_A))\rangle\rangle$

Table 3. (Unicast Time Synchronization) These are the two message formats used for synchronizing the clock of client Alice (A) with that of server Bob (B) according to protocol version 0.3.0.

Name	Direction	Contents
time_request	$A \rightarrow B$:	S_A / N / $h(K_A)$
time_response	$B \rightarrow A$:	S_B / N / $\mathrm{HMAC}[\mathrm{Cook}_B(h(K_A))]\langle S_B \,/\, N\rangle$

ProVerif allows the user to specify so-called "queries", which can be used to specify the required protocol goals (Reference [6, Sect. 2] provides a short introduction to how ProVerif interprets queries). When a goal has been specified in this way, ProVerif can look for a protocol state which violates the condition corresponding to that goal. It can then return one of three possible results:

- There is no state which violates the goal condition.
- A state could be constructed which violates the given goal condition. In this case, ProVerif shows a trace of events leading to that state. The violation of the given condition might represent a viable attack on the protocol but it might also uncover an error in the model underlying the ProVerif code.
- ProVerif cannot prove that the goal is correct but it cannot construct a state violating the goal condition either.

The last case can occur because ProVerif works on an over-approximation of the state space of the input specification. Thus, ProVerif may find a state that violates the goal condition but that is not part of the state space of the input specification. Over-approximation is unavoidable since the decision problem underlying ProVerif is known to be undecidable [10] (however, it is of course semi-decidable). See Reference [6, Sect. 3.3.1] for some more details on the different possible results and how to interpret them, in particular for the third of the above cases.

We now provide an overview of how client and server are modeled in ProVerif. The ProVerif code relevant to the model of the originally analyzed version can be seen in Appendix A). Alternatively, the full code for use with ProVerif can be downloaded under https://www8.cs.fau.de/staff/milius/ProVerif-NTS.rar. The client is modeled via two ProVerif processes:

- The **inner client process** performs, exactly once, the message exchange of the time synchronization phase.
- The **outer client process** runs through the appropriate message exchanges for certificate exchange and cookie establishment exchange in chronological order. When this is done, it ultimately starts a bulk of "infinitely many" iterations[5] of the inner client process running in parallel.

Accurately modeling the server is more complex than modeling the client. This is a consequence of the fact that the server is required to be stateless (except for the global server seed which is client independent). Nevertheless, the server must be able to reply at any time to any of the specified protocol messages, for an arbitrary number of clients. In the model, this is achieved by having one process type for each possible message exchange and running all of these process types in parallel, each with "infinitely many" iterations.

Consequently, the server is modeled via the following ProVerif Processes:

- There are three **inner server processes**, which perform the respective message exchanges for certification, cookie exchange, and time synchronization.
- The **outer server process** just starts "infinitely many" iterations of each of the inner processes, all running in parallel.

6 Results of the First Analysis

In the course of the analysis, two attacks were discovered by checking the protocol goals, listed below as Goals 7–9, with ProVerif. The first discovery is an attack detected by ProVerif mostly due to imprecision in the model underlying the code version c030ut, which was used at the time. This version does not support enough types to distinguish between a single hostname, on the one hand, and a complex structure of arbitrarily many arbitrary data types, on the other hand. The second (and more practically relevant) discovery is a Man-in-the-Middle attack. It exploits the fact that in NTS, as specified in reference [29, version 03],

[5] Having "infinitely many" iterations is what is modeled for the analysis. In practice, this will obviously be a finite number.

a client Alice is given no indication as to whether a cookie has been generated based on her own public key or someone else's.

The goals for this analysis were derived during development of the specification of the protocol, an unusual situation which is due to the fact that the analysis was performed by people who were also authors of the specification. The analysis with ProVerif also involved checks for protocol sanity goals, e. g. that it is possible for the protocol to finish successfully, i. e. with the client accepting some time synchronization data. It additionally involved a weak authenticity goal for the cookie exchange (a weaker version of Goal 9 listed below), but this was only helpful to the extent that it pointed toward the improtance of Goal 8. The ProVerif source code for the queries associated with all of the mentioned goals can be inspected in Appendix A.8. We do not discuss the sanity goals or the weak authenticity goal in more detail, as they do not contribute essential results for the analysis. Instead, we now present the specific security goals directly relevant for this analysis. The first of these goals is the one that represents the full extent of positive security affirmation that our analysis can provide. Both of the attacks that were found lead to a violation of this goal.

Goal 7 (Time Synchronization Authenticity). If Alice accepts the data from a *time_response* message as authentic from Bob, then Bob has indeed sent a *time_response* message with the same time data and the same nonce, secured with the correct cookie for the association between Alice and Bob.

We now present two intermediary security goals which aid with getting more specific information about the nature of the attacks. Both of these goals are necessary preconditions for Goal 7, since the integrity of time synchronization messages depends on the cookie being shared securely between Alice and Bob.

Goal 8 (Cookie Secrecy). If Alice accepts a cookie C from a *server_cook* message as being legitimate, then C is unknown to Mallory.

Goal 9 (Cookie Authenticity). If Alice accepts a cookie C from a *server_cook* message as being legitimate for her association with Bob, then Bob has indeed sent a *server_cook* message in which he has transmitted C and which he has intended for Alice.

We now describe the first of the two attacks that were discovered. This attack violates even the very specific Goal 9, which gives a hint about the underlying weakness: this is not an attack where a legitimate cookie is spied upon, but where a completely new cookie is invented and Alice is deceived into accepting it.

Attack 10. ProVerif discovered an event trace which violated Goal 9 as well as Goal 8. From that trace, the following attack scenario could be derived:

- Mallory intercepts a *client_cook* message CC_1 containing a nonce N.
- She memorizes both values and prevents the message from being delivered in its original form.
- She invents some cookie value X and concatenates it with N.

– She sends a *client_cert* message to the server Bob, which instead of her name M (see the appropriate message diagram for *client_cert* in Sect. 4) contains the concatenated value $\boxed{\begin{matrix} N \\ X \end{matrix}}$.
– The server sends the appropriate *server_cert* message back to Mallory. It contains two values: Bob's certificate, which is irrelevant, and additionally a signature $\mathrm{Sign}[K_B]\left\langle \boxed{\begin{matrix} N \\ X \end{matrix}} \right.$, which Mallory memorizes.
– Mallory then employs concatenation of values known to her as well as encryption under Alice's public key to create a *server_cook* message SC_3 which reads

$$\mathrm{Enc}[K_A]\left\langle \boxed{\begin{matrix} N \\ X \\ \mathrm{Sign}[K_B]\left\langle \boxed{\begin{matrix} N \\ X \end{matrix}} \right. \end{matrix}} \right. .$$

– She sends SC_3 back to Alice, who decrypts it, validates the nonce and signature, and finally accepts the cookie X as the legitimate cookie sent to her from Bob.

As mentioned before, this attack works only if Bob does not recognize when a *client_cert* message contains the concatenation of a nonce and a cookie-length value instead of a hostname. Under this assumption, anyone can get a signature from an arbitrary server for an arbitrary message by abusing the certification message exchange. However, this is unlikely in any practical implementation.

Countermeasure 11. The introduction of proper identifiers for the different message components completely defeats this attack. Successful ProVerif verification, performed on a more strongly typed model, supports this, as at least Goal 9 has been shown to hold there.[6]

In the following we describe the second of the two revealed attacks.

Attack 12. ProVerif discovered an event trace in which Goal 8 (Cookie Secrecy) was violated. From this trace, the following attack scenario could be deduced:

– Mallory intercepts a *client_cook* message CC_1 containing a nonce N and Alice's public key K_A.
– She memorizes both values and prevents the message from being delivered in its original form.
– She creates a new *client_cook* message CC_2 by concatenating the received nonce N with her own public key K_M.
– She sends this newly created *client_cook* message CC_2 to the server Bob.

[6] The reason why Goals 7 and 8 do not hold for this model is that Countermeasure 11 only defends against Attack 10, not against Attack 12.

– Bob sends back the *server_cook* message SC_1, which has the format

$$\mathrm{Enc}[K_M]\left\langle \begin{array}{l} N \\ \mathrm{Cook}_B(h(K_M)) \\ \mathrm{Sign}[K_B]\left\langle \begin{array}{l} N \\ \mathrm{Cook}_B(h(K_M)) \end{array} \right\rangle \end{array} \right\rangle ,$$

which can hence be decrypted by Mallory and which contains a cookie and a valid signature confirming that Bob has created and sent this cookie as an answer to the request with nonce N.
– Mallory intercepts SC_1 and may prevent it from being delivered in its original form. She decrypts it and then instantly re-encrypts it with Alice's public key K_A, resulting in a new *server_cook* message SC_2.
– She sends SC_2 to Alice, who decrypts it, validates the nonce and signature, and finally accepts the cookie $\mathrm{Cook}_B(K_M)$ as the legitimate cookie sent to her from Bob.

As mentioned above, the given attack is only viable due to the following two reasons:

1. On receipt of a *server_cook* message, it is impossible for anyone to know which public key was used as input value for the generation of the given cookie (because of the one-way property of HMAC and the secrecy of the server seed).
2. On receipt of such a message, it is also impossible for anyone to know for whom the message was originally encrypted (since decrypting and then re-encrypting a message leaves no trace).

The reasons above are interdependent, because an honest server like Bob will always encrypt a *server_cook* message with exactly the same public key upon which the generation of the cookie included in that message is based. We provide two possible countermeasures against Attack 12, both supported by ProVerif validations.

Countermeasure 13. The simple addition of the input value used for the cookie generation into the set of values covered by the signature prevents the described attack. This measure was taken for model version 0.3.1 of the protocol. The resulting message format for *server_cook* can be seen in Table 4. With the ProVerif model updated accordingly, Goal 8 and even Goal 7 can be verified to hold.

Countermeasure 14. An alternate countermeasure is to switch the order of encrypting and signing for the *server_cook* message to encrypting first, then signing. This measure was taken for model version 0.3.2 of the protocol. The resulting message format for *server_cook* can be seen in Table 5. Goal 8 and also Goal 7 can then be verified to hold for an accordingly updated model of the protocol.

Table 4. These are the two updated message formats used for the cookie exchange between client Alice (A) and server Bob (B) according to protocol version 0.3.1.

Name	Direction	Contents
client_cook	$A \rightarrow B$:	N / K_A
server_cook	$B \rightarrow A$:	$\mathrm{Enc}[K_A]\langle N$ / $\mathrm{Cook}_B(h(K_A))$ / $\mathrm{Sign}[K_B]\langle N$ / $\mathrm{Cook}_B(h(K_A))$ / $K_A \rangle \rangle$

Table 5. These are the two message formats used for the cookie exchange between client Alice (A) and server Bob (B) according to protocol version 0.3.2.

Name	Direction	Contents
client_cook	$A \rightarrow B$:	N / K_A
server_cook	$B \rightarrow A$:	$\mathrm{Enc}[K_A]\langle N$ / $\mathrm{Cook}_B(h(K_A)) \rangle$ / $\mathrm{Sign}[K_B]\langle \mathrm{Enc}[K_A]\langle N$ / $\mathrm{Cook}_B(h(K_A)) \rangle \rangle$

Countermeasure 14 has some small advantages over Countermeasure 13: it is simpler to include in a specification, and it also has the benefit that an invalid signature can be detected at a slightly lower computational cost, because no decryption operation is necessary in order to obtain the signature value.

7 Conclusion

We have used ProVerif for the analysis of an early version of the NTS specification. We were able to discover considerable weaknesses in draft version 03 of the specification. Those discoveries were considered in the further development NTS and by version 04, the specification was updated [29, Version 04, Sect. 6.3.2] with a measure conforming to Countermeasure 13, which mitigates Attack 12.

In version 05, the usage of the Cryptographic Message Syntax (CMS) [13] was introduced [29, Version 05, Sect. 6] and a new separate document was created for details on how to use this [30, Version 00]. The usage of the CMS conforms to Countermeasure 11 and provides sufficient data type information to help prevent Attack 10. Moreover, the specification has adapted changes in the cookie exchange that match what is described under Countermeasure 14 [29, Version 05, Sect. 6.2.2], [30, Version 00, Sects. 2 and 4.2.2.2]. Overall, these changes prevent both of the attacks presented in Sect. 6, and therefore represent important improvements.

The discoveries are quite common (and, for experts, perhaps obvious) attacks. An important lesson learned in the course of this analysis is: when designing a new security protocol, it is sensible to check its basic structure with a formal tool, even if this tool may not be sophisticated enough to analyze the finer details. A rough formal analysis may already yield valuable results, as even very basic weaknesses may be overlooked when just considering a protocol on paper. Therefore, it can be useful for the analysis to use abstraction in order to dismiss a complex aspect (like timing in our case), as long as the remaining aspects enable a self-contained examination. Furthermore, a question that one might deduce from the analysis is: to what extent could attacks like Attack 10, which are preventable through the use of well defined data types, be relevant in practice? Do protocol developers need to sufficiently consider the question of data types, or is this a matter that should be left to the implementers? It is also noteworthy that we skipped over a deeper evaluation of different model checkers in the course of this analysis, which leaves room for further research comparing the relative merits of different protocol verifiers for the analysis of protocols and draft standards.

For a complete analysis of the full NTS specification, timing and clocks need to be considered (see Sect. 2). Therefore, further work is necessary to properly consider the timing aspect in interaction with security. To this end, ProVerif is not the suitable tool, because it is unable to consider timing and clocks. Model-checkers other than ProVerif (specifically UPPAAL [5] and TAuth [17]) are being considered, but it is yet unclear if they indeed support the considerations of all necessary aspects for a thorough analysis of secure time synchronization techniques. In particular, finding the right approach and tool is important to analyze the security aspects of broadcast time synchronization with NTS, where TESLA is used for authenticity and integrity protection. Using TESLA to secure time synchronization is especially tricky, since TESLA requires rough time synchronization not only initially, but continuously. Hence, successfully disturbing time synchronization just by a small amount might endanger the security of the whole protocol.

We have started a detailed analysis [19] focusing on an intricate attack using delay techniques to compromise time synchronization when it is secured with TESLA-like mechanisms. A draft version of this work is available under https://www8.cs.fau.de/staff/milius/AttackPossibilityTimeSyncTESLA.pdf.

An automated (or semi-automated) formal analysis of the full NTS specification would be beneficial because it could be re-used for other time

synchronization protocols which employ TESLA-like mechanisms, for example the TinySeRSync protocol [31] for wireless sensor networks.

A ProVerif Source Code

In this appendix, we present some of the relevant ProVerif source code used for the preparation of this paper. There have been different code versions involved in the analysis:

- **Code version c030ut:** It models protocol version 0.3.0 but has no dedicated type for hostnames.
- **Code version c030:** It models protocol version 0.3.0 and does have a dedicated hostname type.
- **Code version c031:** It models protocol version 0.3.1.
- **Code version c032:** It models protocol version 0.3.2.

All of the code below is taken from code version c030, which formed the basis of the analysis. Presenting the other versions in full would take up a lot of space, whereas presenting only the differences is difficult. This is because although the changes between the different code versions are minor, they still include numerous lines that are spread far apart. Some comment lines and structuring have been left out for the presentation.

For any reader who is interested in access to the complete code (all code versions, ready for use with ProVerif), it is available under https://www8.cs.fau.de/staff/milius/ProVerif-NTS.rar.

A.1 Cryptographic Primitives

The first lines of the ProVerif source code form the cryptographic primitives that are needed.

```
(* Basics: Keys and Hostnames *)
   type key.
   type hostname.

(* Symmetric Encryption *)
   fun senc(bitstring, key): bitstring.
   reduc forall m: bitstring, k: key;
         sdec(senc(m,k),k) = m.

(* Asymmetric Encryption *)
   type skey.
   type pkey.
   fun sk_of(hostname): skey [private].
   fun pk(skey): pkey.
   letfun pk_of(X: hostname) = pk(sk_of(X)).
```

```
   fun aenc(bitstring, pkey): bitstring.
   reduc forall m: bitstring, k: skey;
         adec(aenc(m, pk(k)), k) = m.

(* Asymmetric Signatures *)
   type sskey.
   type spkey.
   fun ssk_of(hostname): sskey [private].
   fun spk(sskey): spkey.
   letfun spk_of(X: hostname) = spk(ssk_of(X)).

   fun sign(bitstring, sskey): bitstring.
   reduc forall m: bitstring, k: sskey;
         getmess(sign(m,k)) = m.
   reduc forall m: bitstring, k: sskey;
         checksign(sign(m,k), spk(k)) = m.
   letfun signed_message(m: bitstring, k: sskey)
          = (m, sign(m, k)).

(* Hash and HMAC Functions *)
   fun hash(bitstring): bitstring.
   fun keyhash(pkey): bitstring.

   fun seed_of(hostname): bitstring [private].

   fun cookie_gen(bitstring, bitstring): key.
   fun hmac(key, bitstring): bitstring.
```

A.2 Global Variables and Constants

Next we have the declarations of channels and other global variables and constants.

```
(* The Channel for All"Network" Protocol Communications *)
   free c: channel.

(* The Channel and Hostname for Communication with the TA *)
   free ta: channel.
   free TA: hostname.

(* Two Possible Version Identifiers *)
   free version_1: bitstring.
   free version_2: bitstring.
```

A.3 Events

The declarations of ProVerif “events” follow that are used for better traceability of what is happening in what order.

```
(* Server Side Events *)
   event serverError().

   event serverHasOwnCert(hostname, pkey, spkey).
   event serverRespondsTime(bitstring, bitstring,
                            bitstring).

   event serverSaysCookie(key, pkey, hostname).
   event serverGeneratesCookie(key, hostname).

   event serverAcceptsCert(hostname).
   event serverRejectsCert().

(* Client Side Events *)
   event clientError().
   event clientHasOwnCert(hostname, pkey, spkey).

   event clientAcceptsCookie(key, hostname, hostname).
   event clientAcceptsSomeCookie().

   event clientRejectsCookie(key, hostname, hostname).
   event clientRejectsSomeCookie().

   event cookieCompromised(key).

   event clientAcceptsCert(hostname, spkey).
   event clientRejectsCert().

   event client_timereq(bitstring).
   event clientAcceptsTime(bitstring, bitstring,
                           bitstring).
   event clientDiscards(bitstring).

(* Authority Side Event(s) *)
   event authorityGivesCert(hostname, pkey, spkey).
```

A.4 The Trusted Authority Process

The code then moves on to the processes by which the participants are modeled. We first present the process that models the trusted authority. Its only purpose is to generate and distribute certificates for server and client type participants.

```
(* [Process] ::: TRUSTED AUTHORITY ::: [NTS v0.3.0]
   |: Issues certificates on request. *)
   let authority() =
   let skTA = ssk_of(TA) in

    (* The authority receives a certificate request. *)
       in(ta, H: hostname);
```

```
    (* The authority determines the correct public keys
       for the requester. *)
       let pkH = pk(sk_of(H)) in
       let spkH = spk(ssk_of(H)) in

    (* From that information, the authority assembles
       and signs the appropriate certificate. *)
       let certificate = (H, pkH, spkH) in
       let signature = sign(certificate, skTA) in
       let certificate_sig = (certificate, signature) in

    (* The authority sends the response back to the
       requesting participant. *)
       event authorityGivesCert(H, pkH, spkH);
       out(ta, certificate_sig).
```

A.5 The Server Side Processes

Inner Server Processes. Next are those processes that make up the model of the server. The first process represents the server module which deals with the certification message exchange.

```
(* [Process] ::: SERVER CERTIFIER MODULE ::: [NTS v0.3.0]
   |: Replies to a client_cert message with a server_cert
   |: message as specified. *)
   let server_certifier(B: hostname, pkB: spkey,
                        skB: sskey) =

    (* The server acquires the TA's public key. *)
       let pkTA = spk(ssk_of(TA)) in

    (* The server receives a client's certification request. *)
       in(c, X_client_cert: bitstring);

    (* The server extracts the necessary information. *)
       let (version_x: bitstring, A_x: hostname)
           = X_client_cert in

    (* The server requests its certificate chain from the
       trusted authority and performs a validity check on
       the response that it receives. *)
       out(ta, B);
       in(ta, Z_certificate: bitstring);
       let (=B, some_key: pkey, =pkB,
            Z_cert_signature: bitstring)
           = Z_certificate in
       if (B, some_key, pkB)
           <> checksign(Z_cert_signature, pkTA)
       then event serverError()
```

```
    else event serverHasOwnCert(B, some_key, pkB);

      (* The server creates a server_cert response
         as specified. *)
         let msg_server_cert = (A_x, Z_certificate) in
         let msg_server_cert_sign
             = (msg_server_cert,
                sign(msg_server_cert, skB))
             in

      (* The server sends the composed response to the
         requesting client. *)
         out(c, msg_server_cert_sign).
```

The next process involves the server module whose purpose it is to execute the cookie message exchange as well as the required calculations.

```
(* [Process] ::: SERVER COOKIE MODULE ::: [NTS v0.3.0]
   |: Takes a client_cook request, generates the appropriate cookie
   |: and replies with a server_cook message as specified. *)
   let server_cookie(B: hostname, pkB: spkey, skB: sskey,
                     seed: bitstring) =

    (* The server acquires the TA's public key. *)
       let pkTA = spk(ssk_of(TA)) in

    (* The server receives a client's cookie request. *)
       in(c, X_cook: bitstring);

    (* The server matches it to the specified message
       pattern and extracts the necessary information. *)
       let (n_x: bitstring, pkA_x: pkey) = X_cook in

    (* The server builds the cookie for the received client
       (identified via its public [encryption] key pkA. *)
       let cookie = cookie_gen(keyhash(pkA_x), seed) in

    (* It builds the appropriate response *)
       let response = (cookie, n_x) in

    (* It constructs its signature and attaches it to the response. *)
       let signature = sign(response, skB) in
       let response_sig = (response, signature) in

    (* It encrypts it. *)
       let response_sig_enc = aenc(response_sig, pkA_x) in

    (* It sends it back to the client. *)
       event serverGeneratesCookie(cookie, B);
       event serverSaysCookie(cookie, pkA_x, B);
```

```
       out(c, response_sig_enc).
```

Then the server module follows that takes care of the time synchronization message exchange.

```
(* [Process] ::: SERVER TIMESYNC MODULE ::: [NTS v0.3]
   |: Replies to a time_request message with a time_response message as
   |: specified. *)
   let server_time_response(B: hostname, pkB: spkey,
                            skB: sskey, seed: bitstring) =

    (* The server receives a time_request message from a client. *)
       in(c, Y: bitstring);

    (* It extracts the necessary data. *)
       let (t1_y: bitstring, n_y: bitstring,
            pkA_hash_y: bitstring) = Y in

    (* It creates the appropriate time sync data for its response. *)
       new t2: bitstring;

    (* It re-computes the cookie. *)
       let cookie = cookie_gen(pkA_hash_y, seed) in

    (* It composes its response. *)
       let response = (n_y, t1_y, t2,
                       hmac(cookie, (n_y, t1_y, t2)))
           in

    (* It sends its response back to the requesting client. *)
       event serverRespondsTime(n_y, t1_y, t2);
       out(c, response).
```

Outer Server Process. We then see the "outer" server process whose purpose is simply to execute iterations of all the "inner" processes (the modules listed above in Appendix A.5).

```
(* [Process] ::: SERVER GLOBAL PROCESS ::: [NTS v0.3.0]
   |: Executes all server modules at once, running arbitrarily many
   |: instantiations of each of them in parallel. *)
   let server(B: hostname) =

    (* Before running any modules, the server generates an
       unpredictable seed value and remembers its own key pair. *)
       let seed = seed_of(B) in
       let skB = ssk_of(B) in
       let pkB = spk(skB) in

    (* The server then runs all modules. *)
```

```
    !server_certifier(B, pkB, skB)
  | !server_cookie(B, pkB, skB, seed)
  | !server_time_response(B, pkB, skB, seed).
```

A.6 The Client Side Processes

Inner Client Process. Moving on to the client side processes, there is first the "inner" process which takes care of the time synchronization message exchange, including the necessary checks on the MAC.

```
(* [Process] ::: CLIENT TIMESYNC MODULE ::: [NTS v0.3.0]
   |: Generates time_request messages as specified and sends them to a
   |: time server. It then awaits a time_response message on which it
   |: performs the necessary checks as specified. *)
  let client_time_request(A: hostname, pkA: pkey,
    B: hostname, cookie: key) =

   (* The client generates time data and a nonce. *)
      new t1: bitstring;
      new n1: bitstring;

      event client_timereq(t1);

   (* The client constructs its time_request message and sends it. *)
      let request = (t1, n1, keyhash(pkA)) in
      out(c, request);

   (* It receives a time_response message and extracts the necessary
      information. *)
      in(c, X: bitstring);
      let (=n1, =t1, t2x: bitstring, hmacx: bitstring)
          = X in

   (* Depending on the result of validity checks, it either accepts
      the response as authentic or discards it. *)
      if hmacx = hmac(cookie, (n1, t1, t2x))
         then event clientAcceptsTime(n1, t1, t2x)
         else event clientDiscards(X).
```

Outer Client Process. The "outer" client side process follows, which performs the initial message exchanges (server certification and cookie exchange) and then executes instantiations of the inner process.

```
(* [Process] ::: CLIENT GLOBAL PROCESS ::: [NTS v0.3.0]
   |: Executes the steps for association, certification and cookie
   |: exchange, one of each, sequentially. Then it runs arbitrarily
   |: many instances of the client timesync module in parallel. *)
  let client(A: hostname, B: hostname) =
```

```
    let skA = sk_of(A) in
    let pkA = pk(skA) in

    let pkTA = spk(ssk_of(TA)) in

  (* CERTIFICATE PHASE ----------------------------*)

    (* The client sends a client_cert message as specified *)
       let msg_client_cert = (version_1, A) in
       out(c, msg_client_cert);

    (* The client receives a response of type server_cert. *)
       in(c, X_server_cert: bitstring);

    (* The client extracts data from the response. *)
       let (=A, certificate_x: bitstring,
            signature_x: bitstring)
           = X_server_cert in

    (* The client reads the certificate. *)
       let (=B, other_key: pkey, spkB_x: spkey,
            cert_signature: bitstring)
           = certificate_x in

    (* The client performs the necessary test. On failure, it
       exits with an error. On success, the client accepts the
       key given in the certificate as B's public key. *)
       event check();
       if ((B, other_key, spkB_x)
           <> checksign(cert_signature, pkTA))
          || ((A, certificate_x)
              <> checksign(signature_x, spkB_x))
       then event clientRejectsCert()

       else event clientAcceptsCert(B, spkB_x);

         (* COOKIE PHASE ------------------------*)
            let pkB = spkB_x in


         (* The client sends a client_cook message as specified. *)
            new n_cook: bitstring;
            let msg_client_cook = (n_cook, pkA)
                in
            out(c, msg_client_cook);

         (* The client receives a response of type server_cook. *)
            in(c, X_server_cook: bitstring);

         (* It decodes the response and extracts the data from it. *)
```

```
    let X_dec = adec(X_server_cook, skA)
        in
    let ((cookie_x: key, =n_cook),
         signature_x: bitstring) = X_dec
        in

  (* It performs the necessary checks as specified. On success,
    it starts sending time_request messages as specified.*)
    if (cookie_x, n_cook)
        = checksign(signature_x, pkB)
    then event
         clientAcceptsCookie(cookie_x, A, B)
      (* TIMESYNC PHASE ------------------*)
        | !client_time_request(A, pkA,
                               B, cookie_x)
        | ( in(c, =cookie_x);
            event cookieCompromised(cookie_x)

    else event
         clientRejectsCookie(cookie_x, A, B)).
```

Note that the first `else`-branch includes all the code below it. Note also the dedicated listener process given by

```
        | ( in(c, =cookie_x);
            event cookieCompromised(cookie_x)
```

which is started when the client accepts a cookie and does not really represent client behavior according to the protocol, but only listens for the cookie on an open channel. This enables us to check for the loss of a cookie by querying whether the event `cookieCompromised()` is ever executed at all (see Appendix A.8).

A.7 The Environment Process

Here, we present the global ProVerif process which takes care of instantiating all participants.

```
(* [Process] MAIN OVERALL PROCESS ::: [NTS v0.3.0]
   |: Runs everything that needs to be run. *)
   process

   (* More strongly typed version with hostnames,
      would otherwise be"bitstring" type variables *)
      new B: hostname;
      new A: hostname;

   (* There are arbitrarily many clients running, but only one server,
      for simplification in the earlier phase of this analysis. *)
        !server(B) | !client(A, B) | !authority()
      | out(c, A)  | out(c, B) )
```

A.8 ProVerif Queries

Now we present the ProVerif queries. We first consider those queries that concern the cookie exchange.

Sanity – Cookie Exchange. This query makes sure that there is some cookie x which is accepted by the honest client A as coming from the honest server B, i. e. the cookie exchange can be completed successfully.

```
query x: key;
      event(clientAcceptsCookie(x, new A, new B)).
```

This query holds for all four code versions.

Weak Authenticity – Cookie. This query ensures that if the honest client A accepts a cookie x for communication with the honest server B, then B has in fact generated x and released it into the network (note that this gives no guarantee that B intended x for communication with A in particular).

```
query x: key;
      event(clientAcceptsCookie(x, new A, new B))
  ==> event(serverGeneratesCookie(x, new B)).
```

Applying this query to the different ProVerif code versions yields the following results:

- It does **not** hold for code version c030ut. The attack that ProVerif discovers is the first one described in Sect. 6.
- It holds for the code versions c030, c031, and c032.

Authenticity – Cookie. This query strengthens the guarantee acquired with the previous query: it ensures that if the honest client A accepts a cookie x for communication with the honest server B, then B has in fact issued x based on A's public key, and has also encrypted the appropriate message with said public key. This is the query that corresponds to Goal 9.

```
query x: key;
      event(clientAcceptsCookie(x, new A, new B))
  ==> event(serverSaysCookie(x, pk(sk_of(new A)), new B)).
```

The results for this query are as follows:

- It does **not** hold for code version c030ut. Authenticity for the cookie would require weak authenticity for it, which is not given (see above). Also, the Man-in-the-Middle attack described in Sect. 6 works on this version.
- It does **not** hold for code version c030. Again, the corresponding attack is the Man-in-the-Middle attack described in Sect. 6.
- It holds for the code versions c031, and c032

Secrecy – Cookie. This query asserts that if the honest client A accepts a cookie x, then the attacker does not know x. This is realized via the event `cookieCompromised()` from the dedicated listener process as described in Appendix A.6. This query corresponds to Goal 8.

```
query x: key;
      event(cookieCompromised(x)).
```

- This query does **not** hold for code version c030ut and **neither** does it hold for c030: Both of the possible attacks enable Mallory to make Alice accept a cookie that Mallory knows. In the case of the Blind-signature attack she manufactures said cookie herself; in the case of the Man-in-the-Middle attack she maliciously re-distributes a valid key, signed by Bob.
- This query holds for the code versions c031 and c032.

Next, we take a look at some queries that concern the time synchronization message exchange.

Sanity – Time Synchronization. This query checks whether it is possible for the protocol to be run such that the honest client A successfully accepts a timesync response as valid and authentic from the honest server B.

```
query nonce: bitstring, x: bitstring, y: bitstring;
      event(clientAcceptsTime(new A, new B, nonce, x, y)).
```

This query holds for all four code versions.

Authenticity – Time Synchronization. This query ensures that if a timesync message t is accepted by the honest client A as authentic from an honest server B, then B has really issued a message with the exact time data as in t and secured it with the cookie which is generated based on A's public key. This query corresponds to Goal 7.

```
query nonce: bitstring, x: bitstring, y: bitstring;
      event(clientAcceptsTime(new A, new B, nonce, x, y))
  ==> event(serverRespondsTime(keyhash(pk(sk_of(new A))),
                               new B, nonce, x, y)).
```

- As might be expected due to the lack of cookie secrecy, this query also does **not** hold for code version c030ut and **neither** does it hold for c030. Since for these code versions Mallory can gain access to cookies that Alice accepts as valid, she can use those cookies to generate time synchronization packets with maliciously manufactured synchronization information that Alice will accept.
- This query holds for the code versions c031 and c032.

References

1. Abadi, M.: Security protocols: principles and calculi. In: Aldini, A., Gorrieri, R. (eds.) FOSAD 2007. LNCS, vol. 4677, pp. 1–23. Springer, Heidelberg (2007)
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2001, pp. 104–115. ACM, New York (2001). http://doi.acm.org/10.1145/360204.360213
3. Archer, M.: Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In: Workshop on Issues in the Theory of Security, pp. 14–15 (2002)
4. Basin, D., Capkun, S., Schaller, P., Schmidt, B.: Formal Reasoning About Physical Properties of Security Protocols. ACM Trans. Inf. Syst. Secur. **14**(2), 16:1–16:28 (2011)
5. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). http://dx.doi.org/10.1007/978-3-540-30080-9_7
6. Blanchet, B., Smyth, B., Cheval, V.: ProVerif 1.88: automatic cryptographic protocol verifier, user manual and tutorial. Technical report, INRIA Paris-Rocquencourt, 08 2013
7. Blanchette, J.C., Nipkow, T.: Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010). http://dblp.uni-trier.de/db/conf/itp/itp2010.html#BlanchetteN10
8. Cremers, C.J.F.: The scyther tool: verification, falsification, and analysis of security protocols. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008). http://dx.doi.org/10.1007/978-3-540-70545-1_38
9. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Trans. Inf. Theory **29**(2), 198–208 (1983)
10. Durgin, N.A., Lincoln, P., Mitchell, J.C.: Multiset rewriting and the complexity of bounded security protocols. J. Comput. Secur. **12**(2), 247–311 (2004). http://content.iospress.com/articles/journal-of-computer-security/jcs215
11. Ganeriwal, S., Pöpper, C., Capkun, S., Srivastava, M.B.: Secure time synchronization in sensor networks (E-SPS). In: Proceedings of 2005 ACM Workshop on Wireless Security (WiSe 2005), pp. 97–106. ACM, Sept 2005
12. Hopcroft, P., Lowe, G.: Analysing a stream authentication protocol using model checking. Int. J. Inf. Secur. **3**(1), 2–13 (2004)
13. Housley, R.: Cryptographic Message Syntax (CMS). RFC 5652, RFC Editor, September 2009. http://www.rfc-editor.org/rfc/rfc5652.txt
14. IEEE: IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems (2008). http://ieeexplore.ieee.org/servlet/opac?punumber=4579757
15. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, 02 1997. http://www.rfc-editor.org/rfc/rfc2104.txt
16. Levine, J.: A Review of Time and Frequency Transfer Methods. Metrologia **45**(6), 162–174 (2008)
17. Li, L., Sun, J., Liu, Y., Dong, J.S.: TAuth: verifying timed security protocols. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 300–315. Springer, Heidelberg (2014). http://dx.doi.org/10.1007/978-3-319-11737-9_20

18. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 696–701. Springer, Heidelberg (2013)
19. Milius, S., Sibold, D., Teichel, K.: An Attack Possibility on Time Synchronization Protocols Secured with TESLA-Like Mechanisms, Draft version: https://www8.cs.fau.de/staff/milius/AttackPossibilityTimeSyncTESLA.pdf
20. Mills, D., Haberman, B.: Network Time Protocol Version 4: Autokey Specification. RFC 5906, RFC Editor, 06 2010. http://www.rfc-editor.org/rfc/rfc5906.txt
21. Mills, D., Martin, J., Burbank, J., Kasch, W.: Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, RFC Editor, 06 2010. http://www.rfc-editor.org/rfc/rfc5905.txt
22. Mizrahi, T.: Security Requirements of Time Protocols in Packet Switched Networks. RFC 7384, RFC Editor, 10 2014. http://www.rfc-editor.org/rfc/rfc7384.txt
23. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer, Heidelberg (2002)
24. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. J. Comput. Secur. **6**(1–2), 85–128 (1998)
25. Perrig, A., Song, D., Canetti, R., Tygar, J.D., Briscoe, B.: Timed Efficient Stream Loss-Tolerant Authentication (TESLA): Multicast Source Authentication Transform Introduction. RFC 4082, RFC Editor, 06 2005. http://www.rfc-editor.org/rfc/rfc4082.txt
26. Röttger, S.: Analysis of the NTP Autokey Procedures, 2 2012. http://zero-entropy.de/autokey_analysis.pdf
27. Ryan, M.D., Smyth, B.: Applied pi calculus. In: Cortier, V., Kremer, S. (eds.) Formal Models and Techniques for Analyzing Security Protocols, Chap. 6. IOS Press (2011). http://www.bensmyth.com/files/Smyth10-applied-pi-calculus.pdf
28. Saeedloei, N., Gupta, G.: Timed π-calculus. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 119–135. Springer, Heidelberg (2014)
29. Sibold, D., Teichel, K., Röttger, S.: Network time security. Technical report, IETF Secretariat, 07 2013. https://datatracker.ietf.org/doc/draft-ietf-ntp-network-time-security/history/
30. Sibold, D., Teichel, K., Röttger, S., Housley, R.: Protecting network time security messages with the cryptographic message syntax (CMS). Technical report, IETF Secretariat, 10 2014. https://datatracker.ietf.org/doc/draft-ietf-ntp-cms-for-nts-message/history/
31. Sun, K., Ning, P., Wang, C.: TinySeRSync: secure and resilient time synchronization in wireless sensor networks. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, pp. 264–277. ACM, New York (2006)
32. Syverson, P., Meadows, C., Cervesato, I.: Dolev-yao is no better than machiavelli. In: Degano, P. (ed.) First Workshop on Issues in the Theory of Security – WITS 2000, pp. 87–92, Jul 2000. http://theory.stanford.edu/~iliano/papers/wits00.ps.gz