Paper

Security Verification in the Context of 5G Sensor Networks

Piotr Remlein and Urszula Stachowiak

Poznań University of Technology, Poznań, Poland

https://doi.org/10.26636/jtit.2021.153221

Abstract—In order to develop reliable safety standards for 5G sensor networks (SN) and the Internet of Things, appropriate verification tools are needed, including those offering the ability to perform automated symbolic analysis process. The Tamarin prover is one of such software-based solutions. It allows to formally prove security protocols. This paper shows the modus operandi of the tool in question. Its application has been illustrated using an example of an exchange of messages between two agents, with asynchronous encryption. The scheme may be implemented, for instance, in the TLS/DTLS protocol to create a secure cryptographic key exchange mechanism. The aim of the publication is to demonstrate that automated symbolic analysis may be relied upon to model 5G sensor networks security protocols. Also, a use case in which the process of modeling the DTLS 1.2 handshake protocol enriched with the TCP SYN Cookies mechanism, used to preventing DoS attacks, is presented.

Keywords—5G, automated symbolic analysis, Internet of Things, security protocols, sensor networks.

1. Introduction

New communication technologies, such as 5G, are vulnerable to various type of attacks. Devices in such networks deal with huge volumes of seemingly non-valuable data. This means that security rules in such systems are often disregarded by equipment manufacturers, and that users usually do not care about maintaining proper protection mechanisms. The advent of 5G shows that negligence in the implementation of security measures is an enabler of attacks aimed at harming private and public property. Thus, security of 5G-based sensor networks (SN) or IoT systems is an important issue exerting a significant impact on the development of these technologies [1].

To achieve adequate level of security, newly developed systems rely on cryptographic protections that were known previously. However, they are often quite clumsy in terms of design, and the level of security of new solutions is not properly verified. Therefore, it is important to achieve a formal proof of security at the design stage. For this purpose, tools for automatic security verification relying on symbolic analysis are often used [2], such as Tamarin [3].

This paper describes how Tamarin may be used in testing and validation of security protocols. The paper addresses issues related to the security level of 5G sensor networks and the Internet of Things with respect to the characteristics of 5G SN devices.

This paper is structured as follows. In Section 2, a brief description of selected security standards for 5G SN and IoT is presented. Section 3 contains an introduction to Tamarin software. The manner in which this software may be used for protocol security analysis is described, with a simple message exchange between two agents in an IoT or a sensor network, with asynchronous encryption in the transport layer security (TLS) protocol, supporting secure exchange of cryptographic keys, used as an example. Section 4 shows how to search for errors in the model based on the simple example of a Diffie-Hellmann algorithm model. Sections 5 and 6 describe how Tamarin software may be used to validate security protocols based on the DTLS 1.2 handshake. Reference to TLS version 1.2 is given and a model representing migration from TLS-over-TCP to TLS-over-UDP is examined. The Jun Kim model with a modification of the DTLS protocol aimed at adding a collateral TCP SYN-Cookies mechanism [4], [5] is described as well. Datagram TLS seems to be another promising protocol in 5G SN applications, where the broadly understood system resources, such as computational power, operating memory, the number of round-trips necessary to commence the sending of application messages, and the energy used for computing and data transmission, are usually very limited. The summary and conclusions are presented in Section 7.

2. Selected Security Mechanisms for 5G SN and IoT

The limited hardware resources of 5G SN devices often do not allow for the full implementation of typical security mechanisms and advanced cryptographic algorithms. Thus, special security protocols should be developed, such as TLS and datagram transport layer security (DTLS). They are designed to offer specific security rules relied upon while communicating. TLS is a better version of the secure socket layer (SSL) and the terms are often used interchangeably.

TLS utilizes the transmission control protocol (TCP). DTLS is designed for applications whose communications use the user datagram protocol (UDP) and is designed to be no different from TLS. It should also provide a degree of

security that is similar to the one offered by TLS. Because DTLS is based on UDP, communications are unreliable. It is less resource intensive, making it better suited for 5G SN than TLS. The DTLS protocol solves two problems that TLS can experience with datagram transport. In the case of DTLS, it is not possible to use stream ciphers. Therefore, it is possible to decode records from datagrams individually, because the order of delivery of datagrams may differ from the original order. Unlike TLS, it utilizes explicit transport layer messages [6].

These protocols support confidentiality, integrity and authentication at the transport layer. Both of them use asymmetric encryption and the X.509 certification protocol. They use traditional security mechanisms but can also be modified to fit the limited hardware resources of IoT and 5G SN devices [7]. The cryptographic system parameters used during a single session are agreed upon using the TLS handshake protocol. Communication participants can choose the protocol version and the cryptographic algorithms. Optionally, both sides can authenticate each other and define a shared secret key. TLS supports three modes of key agreement: elliptic curve Diffie-Hellman (ECDHE), pre-shared key (PSK), and PSK with ECDHE [8].

The TLS handshake consists of three main phases. The first of them is the key exchange phase, where the keys and cryptographic parameters are established to allow encryption of the transmission. In this phase, the client transmits a *ClientHello* message that includes a random, one-time identifier named "once". The response has the form of a *ServerHello* message that specifies the negotiated cryptographic parameters of the connection. The participants negotiate a shared communication key. Then, in the server parameter exchange phase, other security parameters that are required for authentication are specified. In order to do this, the server sends two types of messages: *EncryptedExtensions* and *CertificateRequest*.

The third and final phase is concerned with server authentication and, optionally, client authentication. The same set of commands is exchanged: *Certificate*, *CertificateVerify*, and *Finished* [8]. The TLS protocol is a hardware resource intensive solution and may lead to overloading the system. It requires additional steps to be taken when setting up communication and calls for significant amounts of memory for storing the certificates. Version 1.3 from 2018 is the most recent iteration of TLS/DTLS. This specification skips many of the outdated cipher algorithms. Version 1.3 is optimized for efficiency and is intended for IoT applications [8], [9].

3. Introduction to Tamarin

To verify the security level of a specific protocol, computational or symbolic analyses are relied upon. Symbolic verification is a relatively quick method that is less detailed than the calculation-based approach. During the process of creating new protocol models, large assumptions are used. Hence, the idea of creating software solutions to increase the level of automation while using this method was conceived. This type of automated analysis has been intensively researched over the past years. Tools of this type use preprogrammed functions with cryptographic primitives and have a predefined model of intruder/attacker behaviors. Their usefulness was proven during the analysis of TLS protocol's version 1.3. Automated symbolic analysis was used at the pre-implementation verification stage. Although this type of analysis is used on a frequent basis, the solutions obtained are characterized by a high level of discrepancy between what can be demonstrated by traditional computational methods and what is offered by fully automatic tools [6], [10].

Tamarin is an open-source software-based solution designed for symbolic analysis and verification of security protocols. It consists of a collection of tools capable of solving many more problems than alternative tools, such as ProVerif, Scyther, and Maude-NPA [4], [11]. Tamarin was designed by the Information Security Group at ETH Zurich. Research is currently underway on new security protocol verification techniques potentially applicable to this tool. Actual security protocols for 5G systems or the IEC 9798 standard are being modeled as well [10], [12], [13].

Tamarin's operation is based on a dedicated language, in which the analyzed protocols are modeled and rules are created for agents – both those involved in information exchange and attackers. The models created may be used for automated generation of proofs related to the analyzed protocols. Proofs are always performed according to similar principles, i.e. a logically justified number of messages exchanged between agents is created for the protocol under investigation. Then, the degree to which these messages are susceptible to a specific attack method is verified. If all known attacks taken into account in the analysis prove to be ineffective, then a symbolic proof of security for the examined instances is obtained. Otherwise, Tamarin provides a counterexample describing the attack.

This type of software performs proofs while working in two modes. The first is fully automated, uses heuristics to find counter-arguments for the performed proofs. This mode, however, does not always allow to obtain an unambiguous answer in the form of a security proof. This is due to the complex nature and numerous message relationships of the investigated model under. The second, interactive mode allows a counterexample to be generated using a graph that includes the agents involved in a given type of attack (Fig. 1). This makes it possible to understand how a specific attack can be carried out. Tamarin was created using the Haskell programming language [10], [12], [13].

During the phase focusing on the protocol's security level, an intruder/attacker model and a communication channel model are created in addition to the symbolic protocol model. The intruder model is based on a set of theorems and algorithms that formalize the knowledge and the capabilities of the attacker. One of the most common models used in Internet security research to describe intruder behavior is the Dolev-Yao model. It is also used in Tamarin. Here,

Fig. 1. Tamarin tool interactive mode.

the attacker can eavesdrop on the transmission, adding information from the message to the fact set. The foe can use knowledge about functions, terms and equations that are not marked as secret. The model also allows the attacker to rely on deduction to derive new facts, and to prepare messages with false facts it is familiar with. One may conclude that the Dolev-Yao model in which the intruder can eavesdrop on information and send modified data to legitimate agents participating in the communication process is a man-in-the-middle type of attack [12]–[14].

Recently, it has been observed that the Dolev-Yao model is not accurate enough for IoT applications. For example, it does not take into account physical attacks which are a common cause of security breaches in 5G SN and IoT networks [14]. Difficulty with modeling rather complex attacker behaviors observed in 5G SN and IoT networks is a major disadvantage as well. Here, the activities of the intruder are not always obvious. The intruder often collects sensitive information and passes it to another agent for exploitation. This makes it necessary to model the intruder's behavior related to specific incidents within sensor networks [14].

Protocols studied with the use of Tamarin are modeled by a set of messages exchanged between agents participating in the communication process. Tamarin does not take into account a scenario in which messages may be lost. It is also necessary to define constraints related to the order in which messages are transmitted using the protocol model.

In the Tamarin model, a specific rule represents the sending of a message and generates an Out(message) fact. At the same time, the message is added to the intruder's knowledge set through the embedded rules. The sending of a particular message is considered complete when the fact Out(message) is transformed into In(message). The occurrence of an In(message) fact models the arrival of the message at the destination [12]. Tamarin's operation is based on a set of translation rules that describe the behavior of the model at the inference stage. The rules specify transitions between states defined in the model of the system or the protocol under analysis. At the inference stage, all potential combinations of the defined rules are analyzed.

A simple model of an exchange of messages between two agents using asymmetric encryption is described below. The code snippet begins with the name of the model that is being proven, in this case the name is "example". The next line is the command that starts the main "begin" block, where custom and built-in functions are defined at the beginning.

The example shows only the built-in asymmetric-encryption mechanisms that are necessary for the model to work:

- p(K) – a function that generates a public key from the private key K,
- aenc(*m*, *pubK*) – asymmetric encryption – a function that encrypts messages *m* with *pubK* key, using an asymmetric encryption algorithm,

- adec(m, K) – an asymmetric decryption function that decrypts an "asymmetrically encrypted" message m with key K.

It is worth noting that K and $pubK$ are chained using the p() function, so that $pubK$ = p(K). Also, if used in the way described above, i.e., with a public key for encryption and private for decryption, the model represents the confidentiality property. There is, although, no reason not to swap $pubK$ and K to encrypt with a private key, thus modeling integrity/undeniability.
In Tamarin, the rules are defined based on the following notation: **[Conditions/facts preceding] -- [Action Facts]-> [Conditions/facts following]**
The main parts of the defined rules are presented in square brackets. The first one refers to the definition of conditions that must be fulfilled to initiate a given rule. The next part is used to define facts representing the given condition, used to prove lemmas. The last part defines the requirements (consecutive facts) which trigger subsequent rules. There may be an optional "let in" fragment in the syntax. This fragment primarily defines variables the use of which should improve code clarity.
In the presented example, the first key distribution rule (*1_distribute_keys*) generates a private key and a public key. The public key so created is then distributed to the two parties involved in the message exchange.

Example 1
```
1. Theory example
2. Begin
3. Builtins: asymmetric-encryption
4. Rule 1_distribute_keys:
5.  [Fr(~privateK)]-->[!Priv($X,~privateK), !Pub($X,
    pk(~privateK)), Out(pk(~privateK))]
6. Rule 2_agent1_sent:
7.  [Fr(~rand), !Pub($agent1, publicK)]--
    [Send($agent1, aenc(~rand, publicK)),
8. Secret(~rand), Role('agent1')]->[Out(aenc(~rand,
   publicK))]
9. Rule 3_agent2_receive:
10.  let received_rand = dec(encrypted_msg,privateK) in
11.  [!Priv($agent2, privateK), In(encrypted_msg)]--[
     Recv($agent2, encrypted_msg),
     Secret(received_rand), Role('agent2')]->[ ]
12. Lemma 1_executable: exists-trace
13.  "Ex agent1 agent2 m #i #j.
   Send(agent1,m)@i & Recv(agent2,m) @j"
14. Lemma 2_secret: all-traces
15.  "All m #i. Secret(m) @i & Role('agent1') @i
      ==> (not (Ex #j. K(m)@j))"
16. End
```

In the description of the rule mentioned above, there is a definition of the Fr() fact that is the random value generated. The next section with facts is omitted because it is empty (square brackets omitted). The last part refers to !Priv() and !Pub() facts, assigning public and private keys to agents 1 and 2, and defines the Out() fact. The Out() fact models the sending of a message through a public channel that is also accessible to attacking agents. In this case, the Out() fact serves the purpose of revealing the public key to the attacker. In order to indicate that it is a permanent fact, it is preceded by an exclamation mark. Such a fact is read, but not consumed, i.e. it does not disappear from the set of available facts when the rule containing it on the left is triggered.
In the next code snippet, the second rule (2_agent1_sent) is defined. This rule accepts the public key, generates a random number and sends it, in an encrypted form, along with the public key. Using the Fr() fact, a random number is generated by the Pub() fact, the public key is accepted, and using the Out() fact, the encrypted random number is sent. The next section of this rule defines the following facts: Send() – the fact reports that *agent1* has sent a message, Secret() – the fact specifies that the value should be secret, Role() – the fact that specifies an identity.
In the (3_agent2_receive) rule, the following facts are defined: In() to receive the message, Priv() to retrieve the private key and adec() function to decode the message. The decrypted message is substituted by the received_rand parameter in the let-in block. The next part of this rule is similar to the previous one, except that the Recev() fact is used to inform that the message was received by *agent2*.
Although this is not shown in the example, it is possible to specify constraints when the rule definitions are presented. The constraints are triggered by the use of action facts. Their purpose is to refine the set of further analyzed protocol usage scenarios, for instance ensuring that *agent1* and *agent2* are always two distinct agents. Then the lemmas that Tamarin will require to be met to prove protocol security are defined. Two lemmas are used to check the feasibility of the model defined in the example and to check whether confidentiality of the transmitted messages is ensured. The first one is used to check if there exists *agent1* that sends a message to *agent2* (whether the model allows, at the very least, communication between agents) and the second one checks if value n, being the secret of *agent1*, has not been disclosed to the attacker, represented by the absence of fact K(n) at any given time j in the public channel. In Tamarin, two modes of lemmas can be identified, namely "exist-trace" and "all-trace". The first requires one condition to be satisfied. The other lemma requires that all conditions be satisfied.
In Tamarin, variables are defined by specific prefixes: $\sim$ denotes a new variable, defined by the Fr() fact, $ denotes a variable that is public and thus does not need to be entered by Fr(), # denotes a temporary variable. The lack of a prefix before a variable name indicates a variable to which the message being sent is rewritten [12].

4. Simple Example of Error Detection

In this section, the Diffie-Hellman (DH) algorithm will be demonstrated, allowing two agents to establish a shared symmetric encryption key without the agents having to transmit any secret data over an untrusted network. A detailed description of the algorithm and its properties can be found in [15]. The presented model is a modification of the solution provided by the Tamarin authors [10], [12].

The aim of the modification was to prove that when adding the second half of the Diffie-Hellman algorithm, the two sides of communication will create the same session key. However, Tamarin responded to such a model by means of a counter example, so this modification demonstrates how the software can be used to identify initially unnoticeable reasoning errors while modeling new protocols. The following section also explains the exact meaning of symbolic analysis in the context of modeling security properties.

At the beginning of the model, information is provided on the types if built-in functions and cryptographic primitives used. For the DH algorithm it will be: "diffie-hellman built-in". Adding this element gives the model access to the cryptographic basis of the DH algorithm – the power operator, "^". As the analysis performed by Tamarin is of the symbolic variety, it only means that:

- the operator symbol "^" has appeared in the model. It has no assigned function,
- symbol "^" satisfies the following relationship: $(a \wedge b) \wedge c = (a \wedge c) \wedge b$.

Due to the fact that Tamarin performs symbolic analysis in the protocol model, it is impossible to determine the values of variables or the implementation of functions. Only their mutual dependencies are defined for functions. The variable values are not considered in Tamarin either. In other programming languages, no different types of variables (terms) exist. In Tamarin, a term can be understood as a symbolic name assigned to a value. Variables in Tamarin, however, have their sorts specifying their properties, mainly in the context of security.

The next step is to define function symbols, the number of arguments for each of them, and information who can use such functions. Again, since Tamarin carries out a symbolic analysis, only a name needs to be defined. There is no way to determine, nor a need to specify what the implementation of the function is. We know as much about the function as follows from the mathematical definition of this concept. The model uses: mac/2, g/0, macKey/0 [private]. Note that if the function is [private], the attacker cannot deduce its value for given arguments, even if he knows all of them. As a result, a 0-argument (i.e. constant) macKey function will remain unknown to the attacker, unless sent unsecured in a network message.

Labeled translation rules constitute another peculiar element. The model proposed in [3] takes into account a half of the process of determining the shared session key, i.e. encrypting the private key by one of the parties, sending it over the network to the other party, and combining the received value with its private key through the other party in order to obtain the session key. In order to enable the parties to authenticate each other, the transmitted message will have a MAC code built on the basis of a secret shared MAC key (macKey constant) known only to the parties, but not to the attacker. The model consists of the following rules shown in Example 2.

Example 2
```
1. //Model of the first step of the DH algorithm. A sends
   //its ''encrypted'' private key to B
2. Rule Step1_A_sends_encrypted_private_key_to_B:
3.  [Fr(threadId:fresh), Fr(privateKeyA:fresh)] --[ ]->
    [Out(<g^(privateKeyA:fresh),mac(macKey,
    <g^(privateKeyA:fresh), A:pub, B:pub>)>),
    SendingAnEncryptedPrivateKey(threadId:fresh,
    A:pub, B:pub, privateKeyA:fresh)]
1. //The second step model of the DH algorithm. B takes
   //A's ''encrypted'' private key and ''encrypts'' it with
   //his private key to obtain the session key.
2. Rule Step2_B_accepts_encrypted_private_key_A:
3.  [SendingAnEncryptedPrivateKey(threadId, A, B,
      privateKeyB:fresh),
4.  In(<encryptedPrivateKeyA, mac(macKey,
      <encryptedPrivateKeyA, B, A>)>)]
5.  --SessionKeyApproval(threadId,
      encryptedPrivateKeyA^(privateKeyB:fresh))]->[]
```

The first rule allows an initiator of communication with a publicly known name *A* to start the process of negotiating the session key with a recipient with a publicly known name *B*. Both identifiers are stored in public variables which do not guarantee the uniqueness of the values at the time of their introduction. To start the negotiation process, initiator *A* obtains a fresh or a secret unique value for the protocol thread identifier (variable *tid*) and a fresh value for variable *x*, representing the private key of *A*. In the same step, *A* "encrypts" its private key by raising constant *g* to the power of *x*, and then places the obtained value in the message.

Occurrence of the *SendingAnEncryptedPrivateKey* fact in a multiset together with a message containing *g^privateKeyA* triggers the second rule. This rule means that if the system state elements are: the *Sending AnEncryptedPrivateKey* fact and a message containing this "encrypted" key, the recipient indicated by the sender can receive the message and process the *g^privateKeyA = encryptedPrivateKeyA* value using his private *keyPrivateKeyB* (which is a fresh variable) as follows: *g^privateKeyA^privateKeyB*.

The recipient assumes that this value is the session key, taking into account that if it swapped the roles with the other side, the other side would get the same key.

In the second rule, the verification of the MAC code message is implicitly written. The requirement for the existence of the In() fact conforms to the fact produced in the first step only if this fact contains a message with the MAC signature based on the constant macKey. This means that the binding of the parameters of the Out() fact generated by the rule: *Step1_A_sends_encrypted_private_key_to_B* and the In() fact required by the rule: *Step2_B_accepts_encrypted_private_key_A* can be understood as rewriting the value from the source variable – a parameter of Out() to the target variable – a parameter of In(). The macKey is the exception here, as it is not a variable, but a constant. Therefore, its value must be the same in the fact consumed by the *Step2_B_accepts_encrypted_private_key_A* rule and in

the fact present in the multiset. This technique is known in computational logic as pattern matching.
The next step is to write down the lemma to specify the security-related protocol properties. The lemma shown as Example 3 states that the attacker has no way of knowing the value negotiated as the session key at any time, even before it is actually agreed upon.

Example 3
```
1. Lemma Session_key_is_never_revealed: all-traces
2. "All #t1 #t2 threadId sessionKey.
     SessionKeyApproval(threadId, SessionKey)
3.   @ #t1 & K(sessionKey) @ #t2 ==> F"
```

The following lemma can be read as such. In every possible trace produced by Tamarin, the condition should be preserved for any moment in time *#t1* and *#t2* and for any *thread id* of *threadId* and any session key *sessionKey*, if the recipient has accepted the session key at any time *#t1* and the opponent knows this key at any time *#t2*, then this is an impossible situation (empty predicate set, logically always equal to untruth).
Since the right side of the implication is always false, the only possibility for the entire implication to be true is if the left side of the implication is false as well. So, each trace produced by Tamarin should conform to Example 4.

Example 4
```
1. "All #t1 #t2 threadId sessionKey. Not
     (SessionKeyApproval (threadId,
2.   sessionKey) @ #t1 & K(sessionKey) @ #t2)"
```

It may be formulated in a much simpler way. In none of the produced scenarios (traces) there is such a value that would be accepted as the session key and would be known to the attacker.
As an experiment, we will try to corrupt the protocol by introducing a rule that will expose a macKey constant over the unsecured network at any time Tamarin finds suitable (i.e., with no time nor causal restrictions), giving the opponent access thereto. The interpretation of this fact may be as follows: knowing the macKey constant, the adversary can spoof the MAC code in any message and, consequently, impersonate the initiator. The recipient, thinking that it is negotiating the session key with the initiator, will accept it after the negotiation. However, this will be the session key established not between the initiator and the recipient, but between the attacker and the recipient. The attacker, as one part to the communication process, knows the session key negotiated, which leads to the falsification of lemma Session_key_is_never_revealed.
The third rule that models such a leak is:

Rule mac_key_reveal: [] -- [macKeyReveal()] -> [Out(macKey)]

It has a blank left side, so it can be executed at any time.
As expected, as a result of the analysis of such a protocol model by Tamarin, we obtain an example of a trace for which the lemma Session_key_is_never_revealed is not true.
Tamarin has thus shown a relationship between the secrecy of two seemingly unrelated protocol elements: the MAC key and the negotiated session key.
Further, a less security-stringent version of the correctness condition can be used by replacing the lemma: Session_key_is_never_revealed by: If_the_mac_key_does_not_leak_before (Example 5).

Example 5
```
1. Lemma If_the_mac_key_does_not_leak_before: all-traces
2. "All #t1 #t2 threadId seassionKey.
     SessionKeyApproval(threadId, sessionKey)
3.   @ #t1 & K(sessionKey) @ #t2 ==> Ex #t3.
   macKeyReveal() @ #t3 & #t3 < #t1"
```

This lemma is constructed so that it is acceptable for the attacker to know the session key, but only if before accepting this key as the session key, the adversary gets to know the macKey (revealing the macKey is equivalent to the macKeyReveal() label in the trace). In this case, Tamarin shows the correctness of the protocol – the fulfillment of the lemma for all possible traces. In practice, this proof can be interpreted, as the macKey must remain secret, but only until the parties determine the session key. Afterwards, its secrecy does not matter.
Here, an attempt will be made to address the problem of implementing only "half" of the DH algorithm. It will be shown that if *B* accepts session key *x* in communication with *A* and *A* accepts session key *y* in communication with *B*, then keys *x* and *y* will be identical.
For this purpose, the first step is to add the *SessionKeyApproval* label informing who accepted the key and in communication with whom. Therefore, it is necessary to modify the *Step2_B_accepts_encrypted_private_key_A* rule so that is assumes the form given as Example 6.

Example 6
```
1. Rule Step2_B_accepts_encrypted_private_key_A:
2.  [SendingAnEncryptedPrivateKey(threadId, A, B,
    privateKeyB:fresh),
3.  In(<encryptedPrivateKeyA, mac(macKey,
       <encryptedPrivateKeyA, B, A>)>)]
4. --[SessionKeyApproval(threadId,
       encryptedPrivateKeyA^(privateKeyB:fresh),
       A, B)]-> []
```

This change does not affect the operation of the protocol, but only the set of information available for the purpose of inference. Therefore, verification of the previously used lemmas is not performed and a new one is introduced – see Example 7.

Example 7
```
1. Lemma Both_sides_get_the_same_session_key: all-traces
2.  "All #t1 #t2 threadId1 threadId2 sessionKey1
    sessionKey2 A B.
3.  (SessionKeyApproval(threadId1, sessionKey1, A, B)
      @ #t1 &
4.  SessionKeyApproval(threadId2, sessionKey2, B, A)
      @ #t2) ==> (sessionKey1 = sessionKey2)"
```

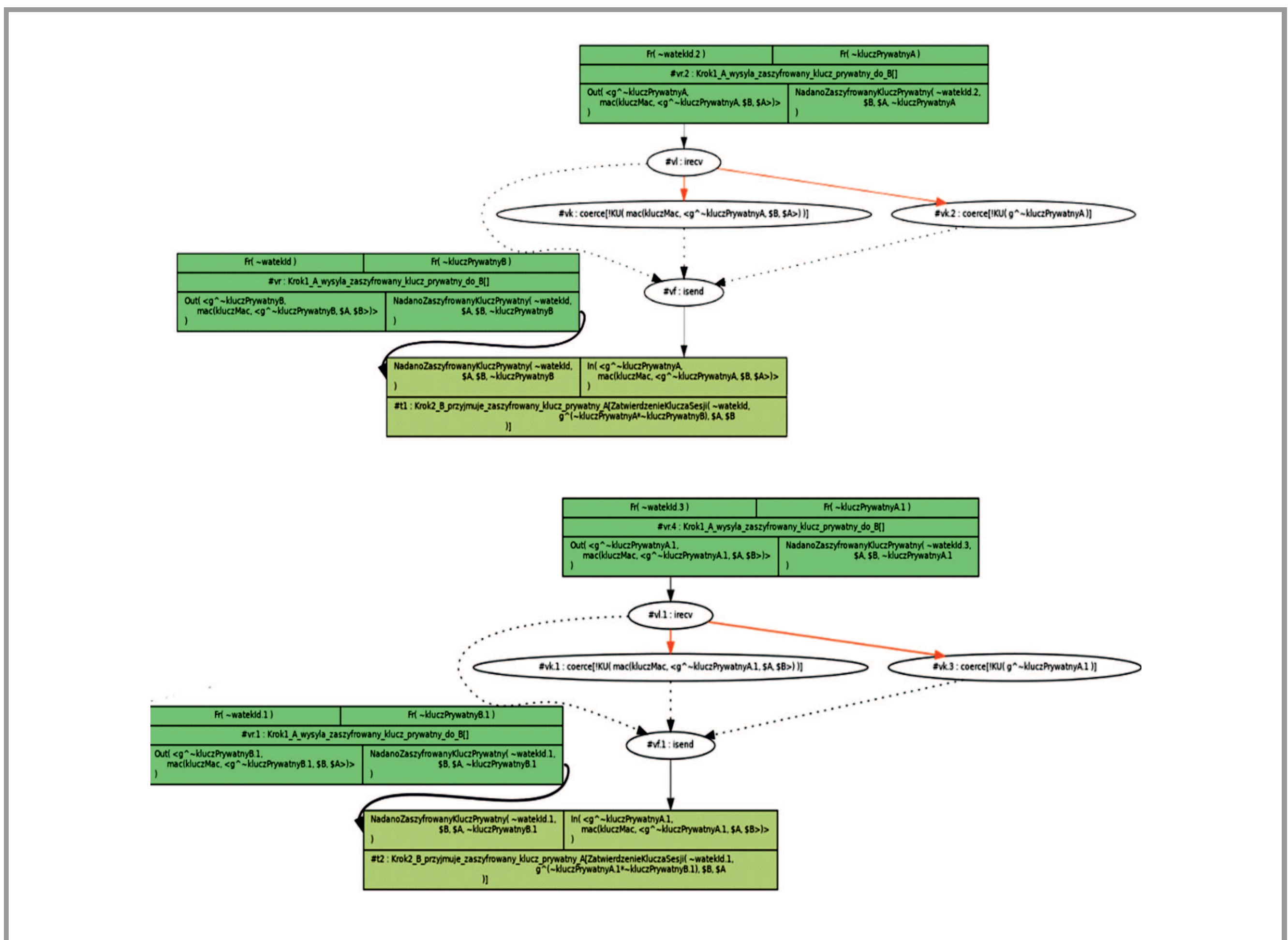


Fig. 2. A trace that does not meet the lemma – visualization of the attack.

It should be read as follows. If any party A has accepted the *sessionKey1* session key negotiated with B (at any time *#t1*) and party B has accepted the *sessionKey2* session key negotiated with A (at any time *#t2*), then both keys are equal. Tamarin is able to construct a counterexample for such a lemma, which means that assembling the "entire use case" of the DH algorithm from "two halves" does not ensure the correct operation of the protocol. In consequence, a situation is feasible where the parties using the same algorithm can generate different session keys.

To read such an attack scenario, it is necessary to examine the trace constructed by Tamarin (Fig. 2).

An explanation of the graphic notation used by Tamarin is required here. In rectangular frames, Tamarin shows the rules used. The notation $x.1$, $x.2$, $x.3$, $x.4$ refers to those variables that appear, in the rules, under the same name x, but have different values in specific rule instances (one rule can be used many times). The ovals show the opponent's actions based on the rules built into the opponent's model in Tamarin. In the trace of the attack scenario, we see four instances of the *Step1_A_sends_encrypted_private_key_to_B* rule: two resulting from the interaction of side A with side B, and two resulting from the interaction of side B with side A. It is worth noting that each time such a rule is applied, the private key of the initiating party is drawn anew (fresh variable). So, we have two sides A and B and four private keys: x, $x.1$, y and $y.1$. Either because of the attacker or because of a network property that does not keep the order of messages, sessions start to intertwine. As a result, party B (peer of the first session) accepts the session key from its first private key (y) and the second private key $A(x.1)$, while party A (peer of the second session) accepts the session key generated from its first private key ($x.1$) (x) and the second private key $B(y.1)$. On the A side, the $g{}^\wedge y{}^\wedge x.1$ session key is created and on the B side – $g{}^\wedge x{}^\wedge y.1$. Even if the ,$\wedge$ operator is used, two session keys will exist: $g{}^\wedge x{}^\wedge y.1$ and $g{}^\wedge x.1{}^\wedge y$, and since x and y are fresh, this guarantees that values x, $x.1$, y, and $y.1$ are different. Consequently, keys $g{}^\wedge x{}^\wedge y.1$ and $g{}^\wedge x.1{}^\wedge y$ accepted as session keys by B and A, respectively, are different. This is in contradiction to the lemma.

This problem can be solved by limiting the application of the *Step1_A_sends_encrypted_private_key_to_B* rule to the same initiator A and communication partner B at any given time. In other words, A is allowed to initiate negotiations with B only once in the entire trace. To do this, a label needs to be added first to the rule shown in Example 8.

Then, an auxiliary mechanism is relied upon to determine which traces generated by Tamarin can be considered worthy of further analysis. This mechanism is called a restric-

Example 8
```
1. Rule Step1_A_sends_encrypted_private_key_to_B:
2.   [Fr(threadId:fresh), Fr(privateKeyA:fresh)]--
3.   [StartEstablishingSessionKey(A:pub, B:pub)]->
     [Out(<g^(privateKeyA:fresh),
       mac(macKey, <g^(privateKeyA:fresh), A:pub,
       B:pub>)>),
4. SendingAnEncryptedPrivateKey(threadId:fresh,
       A:pub, B:pub, privateKeyA:fresh)]
```

tion. In this case, the restriction should allow only one label named *StartEstablishingSessionKey* to be present in the trail for a given initiator *A* and peer *B* (Example 9).

Example 9
```
1. Restriction Only_one_session_between_A_and_B:
2.   "All A B #t1 #t2. (StartEstablishingSessionKey(A, B)
       @ #t1 &
3.    StartEstablishingSessionKey(A, B)  #t2) ==> #t1
        = #t2"
```

Constraints are constructed similarly to lemmas. The above limitation can be understood as follows: for any initiator *A* and its peer *B* and for any moments in time *#t1* and *#t2*, if at time *#t1* and at time *#t2* there is a label *StartEstablishingSessionKey*(*A*, *B*) in the trace, *#t1* and *#t2* are always the same moment. For such a supplemented protocol model, Tamarin is no longer able to produce an attack scenario. Examples of the implementation of common restrictions can be found in [12].

5. Model of DTLS 1.2 Handshake

In this section, the use of the Tamarin prover to model security protocols and to verify their correctness is presented. The analysis concerns the DTLS 1.2 handshake protocol with the optional TCP SYNCookies mechanism modeled by the authors on the basis of documentation [19] and models provided by the Tamarin tool developers and Jun Kim [3]–[5].

In TLS/DTLS, a handshake is a step that is meant to negotiate symmetric encryption-decryption keys by the parties to the connection, one for each of them, without any confidential information being transmitted over an unsecured connection. The use of symmetric cryptography as a means of securing the connection and asymmetric cryptography only as a means of securely establishing symmetric keys stems from significant differences in the speed of operation and, hence, the device load. Asymmetric algorithms are, in general, more resource-demanding, but they guarantee a higher level of security. Such a mixed approach is especially appealing to applications in resource-constrained IoT environments.

The TLS handshake, in both version 1.2 and 1.3, has a certain disadvantage that is impossible to leave out when discussing IoT applications. These mechanisms are security measures for TCP connections established at the transport layer and both rely on the properties of this protocol. The use of stream ciphers for security-related data is only possible if the lower layer guarantees the delivery of packets in the exact order in which they were sent. Effective execution of the handshake in a short time frame will only be possible if the lower layer ensures the retransmission of those messages for which no confirmation has been provided in over an extended period of time. On the other hand, as far as the transport layer is concerned, protocols within the TCP/IP stack, UDP may be particularly interesting for IoT devices. While TCP requires the IoT device to store the connection and application state in its memory, UDP requires connection the state-related information only. At the same time, preventing the datagrams from being lost and reordered does not seem too complicated, provided that it is necessary at all. Therefore, the use of UDP reduces the amount of system resources used, being a big advantage of a protocol designed to work on IoT devices. It should also be emphasized that saving one round trip time per handshake when switching from TLS 1.2 to TLS 1.3 is a relatively poor gain in a scenario where it is always necessary to establish the TCP connection first when performing another 3-way TCP handshake.

The DTLS protocol is a solution that combines the security-related advantages of TLS with the simplicity of UDP. This protocol was created by extending TLS, i.e. allowing it to function properly on a transport layer that uses datagrams instead of connections. The essence of such a solution is to provide a mechanism that is as similar to TLS as possible, but copes with the characteristics of datagrams, i.e. the potential of datagrams getting lost in the network and changing the order of their delivery. DTLS compensates for the deficiencies of the UDP protocol only when performing a handshake. In DTLS handshake messages, there is a field for a sequence number, similar to that in TCP packet headers. DTLS also introduces timers responsible for measuring the time provided for the response of the other party.

When a predetermined value is reached, the last sent message is retransmitted, since either the message, or the response to it has been lost. Additionally, due to the fact that a single authentication certificate for one or both communication sides does not fit into a single datagram, DTLS supports protocol message fragmentation and reassembly at the receiver. In addition to establishing a DTLS "connection", i.e. after negotiating the session keys, the protocol maintains the properties of datagrams while handling application traffic.

As a result, the data that the applications send through the now-secured communication channel is encrypted in datagrams and is sent similarly to normal unsecured traffic. Encrypted datagrams can also get lost and can change their order during a transmission. The task of dealing with these anomalies is handled, however, by the application. DTLS only needs to ensure that the receiver can perform the decoding operation correctly, regardless of any missing and reordered datagrams. This is done by completely abandoning the use of stream ciphers in favor of stateless ciphers.

In addition, in order not to raise any alarms in an overzealous manner, thus unnecessarily breaking a secure session in the event of receiving a message that is not successfully decoded, DTLS adds an epoch number to each message that encapsulates the application data. The epoch is the period over which the same session keys are valid. The re-negotiation of these keys serves as a boundary of such an epoch. So, if a message sent by the client ending the handshake is overtaken, for example, by a message with the application data sent later, the server-side protocol may consider such a situation safe, because it has the ability to decrypt the application data (the handshake has already ended), and the epoch number in the decrypted datagram is greater, by one, than during the handshake. A similar situation occurs with the assumption that session keys are changed periodically and that a limited number of packets or bytes can be sent with a single key.

In the presented DTLS 1.2 handshake model, aspects related to mechanisms used for compensating for loss and reordering of datagrams may be considered as working correctly and securely because they are equivalent to the mechanisms known from TCP. Epoch numbers, in turn, are a mechanism used, in particular, during the exchange of application data. It follows that the DTLS handshake model prepared for Tamarin should be very similar to the TLS handshake model if the same protocol versions are compared.

The DTLS handshake model proposed in [3] is, in fact, similar to the TLS handshake model [5]. The first difference is that each handshake message is labeled with a HMAC signature. We model this signature by introducing a 1-argument HMAC function symbol unbound by any equality features. The attacker can compute the hash knowing all the required arguments, but cannot deduce arguments knowing the hash only. The TLS model equivalents of C_1 and S_1 rules will therefore initially look as shown in Example 10.

Example 10

```
1. Rule C_1:
2.   [Fr(~nc), Fr(~sid)]--[]->[Out(<$C, ~nc, ~sid, $pc ,
       HMAC(<$C, ~nc, ~sid
3.       $pc>)>), St_C_1($C, ~nc, ~sid, $pc)]
1. Rule S_1:
2.     [In(<$C, nc, sid, pc, HMAC(<$C, nc, sid, pc>)>),
         Fr(~ns)]--[]->[Out(<$S, ~ns, sid, $ps>),
3.         St_S_1($S, $C, sid, nc, pc, ~ns, $ps)]
```

However, this model does not take into account one key aspect: the mechanism of preventing DoS attacks on IoT devices. A ready-made mechanism of this type, TCP SYN Cookies [18], can be used when the security protocol is based on the TCP transport layer. In the case of DTLS, a similar mechanism must be built into the security protocol. The specification states that implementation of this mechanism is optional. On the other hand, defense against DoS-type attacks in the case of an IoT network is a problem of such great importance that the mechanism in question should definitely be taken into account in the model. It is so due to the fact that a much greater number of "small" devices is expected to be present in an IoT network than in a typical computer network.

The solution proposed in [19] introduces a preliminary step into the protocol, preceding the sending of the *ClientHello* message by the client. Its task is to inform the server about the intention of establishing a secure session, without the server having to consume any resources. The server authorizes such an attempt to start a conversation by placing a cookie – a hashed value that contains connection parameters and a secret key known to the server only. Then, the client which acts, after receiving the answer, according to the standard procedure, confirms its authorization by attaching the previously received cookie to the *ClientHello* message. Only after receiving such a message and after verifying the correctness of the cookie, does the server allocate resources for the purpose of creating a secure session.

If it is only the server that allocates resources as a result of a complex client interaction, the attacker must maintain a sufficiently large number of fully functional malicious clients in order to consume all server resources. It may turn out to be unprofitable, unlike in a situation in which such protection is not used. Then the attacker would only need to craft an appropriate number of malicious *ClientHello* messages.

Adding a preliminary step to the model [5] requires defining two additional rules: C_0 and S_0.

The client sends the *ClientHello* message without a cookie. The server does not need to support the DoS protection mechanism, so it can respond immediately with the *ServerHello* message, and it can also request additional verification in the form of *ClientHelloVerify* – see Example 11.

Example 11

```
1. Rule C_0:
2.  [Fr(~nc), Fr(~sid)]--[]->
    [Out(<'client_hello', $C, $S,
    ~nc, ~sid, $pc,
3.    HMAC(<'client_hello', $C, $S, ~nc, ~sid, $pc>)>),
      St_C_0($C, $S, ~nc, ~sid, $pc),
4. //The client in this state can receive the ServerHello
   //directly St_C_1($C, $S, ~nc, ~sid, $pc),
5.    CookieFreeSession(~sid)
6. /* No match in the protocol; the server will decide
   /* whether or not to use cookies in this session by
   /* deleting or not deleting this fact when receiving the
   /* client_hello message
7. */]
```

The server can respond to *ClientHello* without a cookie by sending *ClientHelloVerify* with a freshly generated cookie. This decision is to be made by the server once per handshake (Example 12).

There are some further improvements introduced to the rules presented above. As a result of constant evaluation of the evolving DTLS 1.2 handshake model and based on the assessment of the produced counter-examples, the decision was made to:

- include both sender information and receiver information in each message to be sent,

Example 12

```
1. Rule S_0:
2.  Let cookie = h(<C, S, sid , ServerSecret(S)>) in
3.    [In(<'client_hello', C, S, nc, sid, pc,
        HMAC(<'client_hello', C, S, nc, sid, pc>)>),
4.    CookieFreeSession(sid)]--[ AssignsCookie(sid, S,
        C, cookie)]->
5.    [Out(< 'client_hello_verify', S, C, sid, cookie,
        HMAC(<'client_hello_verify', S, C,
6.          sid, cookie>)>)]
```

- include, in the argument, client and server state facts, not only session identifiers and sids, but also information about the other party,
- introduce the *CookieFreeSession*(*sid*) fact to ensure a constant decision on whether or not to use the cookie throughout the entire handshake. It is pointless for the server to first require the cookie and then resign from its verification.

Furthermore, due to the fact that the messages from the first stage of the handshake (*ClientHello* in the variant with or without cookies and optional *ClientHelloVerify*) require the type of message to be specified (the first field in the message containing a descriptive constant), the rules responsible for receiving them also need to be modified. The same applies to messages exchanged later.

The client sends *ClientHello* again, this time with a cookie. The condition for this rule to work is that the client has previously sent *ClientHello* without the cookie. So, there is a state from which it can recover the connection parameters, see Example 13.

Example 13

```
1. Rule C_1:
2.  [In(<'client_hello_verify', S, C, sid, cookie,
        HMAC(<'client_hello_verify',
          S, C, sid, cookie>)>),
3.  St_C_0(C, S, nc, sid, pc)]--[ResendsCookie(sid, C, S,
        cookie)]->
4.  [Out(<'client_hello_with_cookie', C, S, nc, sid, pc,
        cookie,
5.      HMAC(<'client_hello_with_cookie', C, S, nc,
          sid, pc, cookie>)>),
6.  St_C_1(C, S, nc, sid, pc)]
```

A rule that models the server's response to *ClientHello* with a cookie (Example 14).

Example 14

```
1. Rule S_1:
2.  Let cookie = h(<C, S, sid, ServerSecret(S)>) in
3.  [In(<'client_hello_with_cookie',;C, S, nc, sid, pc,
      cookie,
4.     HMAC(<'client_hello_with_cookie', C, S, nc,
         sid, pc, cookie>)>), Fr(~ns)]--
5.  [VerifiesCookie(S, C)]->
        [Out(<'server_hello', S, C, ~ns, sid, $ps
6.          HMAC(<'server_hello', S, C, ~ns,
              sid, $ps>) >),
7.        St_S_1(S, C, sid, nc, pc, ~ns, $ps)]
```

The *ServerSecret*/1 function symbol annotated as [private] has been introduced to generate the cookie. It returns, for each server, a secret value, and since it is a private symbol, the opponent cannot know this value for any of the servers appearing in the trace. That is, of course, if the value for the server is not sent directly over an unsecured network at any time. The second and third steps of the handshake remain generally unchanged, except for adding descriptive constants to messages, as well as sender and receiver identifiers to both messages and state-facts. It is also crucial that the rules have been supplemented with additional labels that will allow a conclusion on the use and verification of the cookie to be made later: Assigns Cookie(who, to_whom), ResendsCookie(who, to_whom) and VerifiesCookie(who, from_whom). The labels specified above are related to generating and assigning a cookie to the connection, sending the received cookie back to the server and verifying it by this server, respectively.

In order to best adapt the model described here to the requirements of the DTLS 1.2 specification, the possibility of the server deciding not to use the *ClientHelloVerify* mechanism was introduced. The server then responds to the *ClientHello* message without a cookie directly with the *ServerHello* message. To model this, an alternative version of the *S*_1 rule was introduced, known as *S*_1_no_cookie (Example 15).

Example 15

```
1. Rule S_1_no_cookie:
2.  [In(<'client_hello', C, S, nc, sid, pc,
        HMAC(<'client_hello', C, S, nc, sid,
3.    pc>)>), Fr(~ns), CookieFreeSession(sid)]--[]->
    [Out(<'server_hello', S, C,
4.     ~ns, sid, $ps, HMAC(<'server_hello', S, C, ~ns,
          sid, $ps>)>),
5.  St_S_1(S, C, sid, nc, pc, ~ns, $ps)]
```

6. Security of DTLS 1.2 Handshake

The model designed in the manner described above, based on the proof performed by Tamarin, is equivalent in terms of security to the TLS 1.2 handshake and TLS 1.3 handshake protocols, if the same lemmas are used for comparison.

The Meier's model [3] defines three lemmas that check the correctness-related properties of the TLS 1.2 handshake. The first lemma requires that both *keyS* and *keyC* session keys be secret, both from the client's and the server's point of view. In the Tamarin syntax, this is stated in the manner presented in Example 16.

Example 16

```
1. Lemma DTLS_session_key_secrecy:
2.  "not(Ex S C keyS keyC #k.SessionKeys(S, C, keyS,
      keyC) @ k & ( (Ex #i.
3.    K(keyS) @ i) | (Ex #i. K(keyC) @ i)) & not (Ex
    #r. RevLtk(S) @ r) & not (Ex #r. RevLtk(C) @ r)
4.    )"
```

The above notation can be interpreted that it is impermissible for both communication parties *S* and *C* to generate a pair of session keys that would be known at any stage of the task execution process by an attacking intruder without leaking the private key of *S* and leaking the private key of *C*. This property allows us to maintain four common high-level security features: confidentiality, integrity, availability and non-repudiation.

The second lemma (Example 17) is used to model the behavior of injective consensus ownership. This condition allows for the unconditional possibility of establishing a secure connection even when the adversary is able to prepare malicious messages and send them over the network.

Example 17
```
1. Lemma injective_agree:  all-traces
2.  "All sid actor peer params #i. Commit(sid, actor,
       peer, params) @ i ==>
3.  (Ex #j. Running(sid, actor, peer, params) @ j & j <
       i & not(Ex actor2 peer2 #i2.
4.       Commit(sid, actor2, peer2, params) @ i2 &
           not(#i = #i2))) (Ex #r. RevLtk(actor) @ r) |
5.       (Ex #r. RevLtk(peer) @ r)"
```

The lemma from Example 17 can be interpreted as follows: if an actor claims that it can send application messages via a secure channel to the peer, then such a peer had to be seen before as working (started a handshake) with identical parameters and, in addition, there is no other pair of *actor2* and *peer2* that would use the same connection parameters (session keys). The process of ensuring that *actor != actor2* and *peer != peer2* is performed by searching for Commit(*actor2*, *peer2*, ...) at a different moment than the one at which Commit(*actor*, *peer*, ...) appeared. The "!=" operator means "not equal". All of the above conditions must be fulfilled, unless there has been a leak of an actor's private key or a peer's private key.

The last lemma (Example 18) was introduced for the purpose of assessing the feasibility of the protocol, and therefore its correctness, not necessarily in terms of security. It requires the ability to establish a secure connection while satisfying all conditions defined at the rule level, without revealing keys.

Example 18
```
1. Lemma DTLS_session_key_setup_possible:  exists-trace
2.  "(All x y #i. Eq(x,y) @ i ==> x = y) & (Ex S C keyS
       keyC #k. SessionKeys(S, C, keyS, keyC) @ k &
3.     not (Ex #r. RevLtk(S) @ r) & not (Ex #r. RevLtk(C)
       @ r)
4.     )"
```

This lemma can be interpreted as follows. There is at least one possibility that any communication party *C* can negotiate session keys *keyS* and *keyC* by communicating with party *S* without revealing the private key of party *S* or the private key of party *C*. Additionally, the combination of two elements labeled Eq() must always imply equality between them. Tamarin makes it possible to show that all three lemmas are satisfied for the Meier model. In addition, it is necessary to introduce one more lemma to test the correctness of the *ClientHelloVerify* mechanism (Example 19).

Example 19
```
1. Lemma
server_accepts_connections_only_from_clients_with_valid
_cookie: all-traces
3.  "All sid S C Cparams Sparams #t0 #t2 #t3 cookie.
4.     (Commit(sid, S, C, Sparams) @ #t2 &
         Commit(sid, C, S, Cparams) @ #t3 &
         AssignsCookie(sid, S, C, cookie) @ #t0)
5.     ==> Ex #t1. (VerifiesCookie(sid, S, C, cookie)
          AssignsCookie(sid, S, C, cookie) @ #t0)
             (@ #t1 & (#t1 < #t2) & (#t1 < #t3)
6.           & (#t0 < #t1))"
```

It can be read as follows. Each server in communication with any client can confirm the successful establishment of a secure session, which is also confirmed by the client, but only on condition that it previously received a valid cookie from that client.

The fulfillment of this lemma could not be proven at first. Although no evidence of Tamarin looping while in operation was observed, Tamarin always exhausts its entire allocated RAM while producing a proof.

Hypothetically, this lemma is non-provable under the researched conditions due to increased model complexity which stems from the server being allowed to choose whether or not to use the cookie. To limit the degree of complexity, one of the choices may be forced, with the use of the cookie being the preferred solutions. Such an extortion could be introduced into the model by means of a imposing a constraint on the "correct" trace, i.e. the trace that is subject to further analysis, requiring that each server uses a cookie at least once in every conversation with the client – see Example 20.

Example 20
```
1. Restriction server_required_to_use_cookies:
2.  "All S C MS Skey Ckey #t2. Running(S, C,
      <'server', MS, Skey, Ckey>) @ #t2
3.     ==> Ex #t1. VerifiesCookie(S, C) @ #t1"
```

Additionally, a counter-example encountered during the model development phase shows that a constraint is needed in which the operations of sending and verifying cookies between server *S* and client *C* are possible only when *C* != *S*. In other words, the execution trace is subject to further analysis, provided that no client is present in it as a server in the same session – see Example 21.

Example 21
```
1. Restriction no_self_session_when_running:
2.  "All sid S C params #t. Running(sid, S, C, params)
3.  @ #t ==> not(S = C)"
```

However, we decided to take a more radical path and divide the presented model into two branches. One with the server never using a cookie, and the other unconditionally requiring the server to do so. Such an approach is practically justified, since we do not see any reason for the server

accepting the DTLS "connection" to allow a no-cookie and cookie handshake simultaneously.
The cookie-free solution is created by commenting the *S*_0 rule, which allowed the server to send the *Client HelloVerify* message. This solution meets the first three lemmas. It is clear that the fourth lemma, server_accepts_connections_only_from_clients_with_valid_cookie, is always fulfilled for the no-cookie model, as the left side of the implication can never be true.
Instead, a cookie-enabled solution is created by commenting the *S*_1_no_cookie server rule, allowing it to respond with a *ServerHello* message to a *ClientHello* message without a cookie. In addition, the client's ability to move from the "*ClientHello* sent" state to the "*ServerHello* received" state was blocked. This solution meets the first three lemmas. In the case of the fourth lemma, it produces a counterexample which can be understood as follows:

- Client *C* starts a handshake with server *S*, thus initiating session *sid*. During that handshake, the *ServerHello* message is spoofed. The client receives a *ServerHello* message from an attacker with maliciously planted cryptographic parameters. This leads the client to believe that the handshake with the impostor may be continued and, as a result, the client claims the establishment of DTLS session keys with the host it considered to be *S*.
- Much later, server *S* receives a valid *ClientHelloWithCookie* message and claims it has verified the cookie in session sid with client *C*. This is a violation of the fourth lemma's time constraints. In parallel, there is another handshake in progress between server *S* and client *C*, using session identifier *sid.1*. Note that the attacker has enough information sniffed during the aforementioned handshake to spoof the *ClientHandshakeFinished* message of the current handshake in order to make it look like part of session *sid*. Also, server *S* is still waiting to finish the handshake with client *C* as part of session sid. As a result, the attacker can lead the server to believe it has negotiated a secure session sid with *C*, which is a prerequisite of the fourth lemma. And because client *C* claims establishing a secure session sid before the server verifies the cookie, lemma four is not fulfilled.

However, the attacker's actions are enough for the client to establish a safe session with the attacker, with the latter thinking he is talking to the server. The attacker can now position himself between the client and the server using the classic man-in-the-middle attack. This is in line with the proposed protocol model, as it examines the possibility of leakage of session keys and not the possibility of the attacker planting their own. In practice, protocols such as DTLS are secured against MITM with the help of PKI – requiring the server at least to have a certificate verifiable by an external, trusted oracle (a certificate authority). This is beyond the scope of the presented model.
It is worth noting that we are defending the server from DoS attacks, and the attacker's (malicious client's) actions have nothing to do with simplicity. Especially they cannot be executed if the client is to be stateless. As this is the only counterexample (attack scenario), the protocol analysis ends with the conclusion that with the exception of the MITM scenario (not considered in terms of security and producing the only counterexample in terms of cookies), the modeled DTLS protocol retains its security properties and the cookie mechanism works correctly. Based on the evidence, the DTLS 1.2 handshake with the additional *ClientHelloVerify* mechanism is a security equivalent of the TLS 1.2 handshake and the TLS 1.3 handshake.

7. Conclusions

There is no doubt that guaranteeing the appropriate level of security in 5G SN or IoT networks is an important requirement for the reliability of these networks.
Research focusing on improving the security of existing solutions, as well as on ensuring new and secure types of connections between devices operating within 5G SN or IoT networks, and on ensuring fully secure services rendered with the use of such networks, is ongoing. The deployment of new solutions involves the creation of new security protocols. In order to automate the process of checking correctness of the security protocols proposed, relevant software tools are created, such as Tamarin.
This paper presents how an automatic symbolic analysis tool can be used at the design stage to perform the security analysis and to verify the correctness of operation of newly proposed protocols used in 5G SN or IoT environments, as well as in other modern sensor networks.

References

[1] M. Nadimpalli, "Internet of Things – future outlook", *Int. J. of Innov. Res. in Comp. and Commun. Engin.*, vol. 5, no. 6, 2017 [Online]. Available: https://www.rroij.com/peer-reviewed/internet-of-things-future-outlook-85898.html

[2] S. Helme, "Perfect forward secrecy – an introduction", 2014 [Online]. Available: https://scotthelme.co.uk/perfect-forward-secrecy

[3] Tamarin prover [Online]. Available: https://github.com/tamarin-prover/tamarin-prover/blob/develop/examples/classic/TLS_Handshake.spthy

[4] J. Y. Kim, R. Holz, W. Hu, and S. Jha, "Automated analysis of secure Internet of Things protocols", in *Proc. of the 33rd Ann. Comp. Secur. Appl. Conf. ACSAC 2017*, Orlando, FL, USA, 2017, pp. 238–249 (DOI: 10.1145/3134600.3134624).

[5] J. Y. Kim, Automated-security-verification-of-IoT-protocols [Online]. Available: https://github.com/jun-kim/Automated-security-verification-of-IoT-protocols/blob/master/CoAP_DTLShandshake.spthy

[6] T. Cole, "Interview with Kevin Ashton – inventor of IoT: Is driven by the users", Smart Industry the IoT Business Magazin, 2018 [Online]. Available: https://www.smart-industry.net/interview-with-iot-inventor-kevin-ashton-iot-is-driven-by-the-users/

[7] T. Salman and R. Jain, "A survey of protocols and standards for Internet of Things", *Adv. Comput. and Commun.*, vol. 1, no. 1, 2017 (DOI: 10.34048/2017.1.f3).

[8] E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.3, draft-ietf-tls-tls13-28 - 20, 2018 [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-tls13-28

[9] E. Rescorla, H. Tschofenig, and N. Modadugu, The Datagram Transport Layer Security (DTLS) Protocol Version 1.3, draft-rescorla-tls-dtls13-01-13, 2017 [Online]. Available: https://datatracker.ietf.org/doc/html/draft-rescorla-tls-dtls13-01

[10] S. Meier, B. Schmidt, C. Cremers, and D. Basin, “The Tamarin prover for the symbolic analysis of security protocols”, in *Computer Aided Verification 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, N. Sharygina and H. Veith, Eds. *LNCS*, vol. 8044, pp. 696–701. Springer, 2013 (ISBN: 9783642397981).

[11] J. Thakkar TLS 1.3 Handshake: Taking a Closer Look, 2018 [Online]. Available: https://www.thesslstore.com/blog/tls-1-3-handshake-tls-1-2/

[12] D. Basin, C. Cremers, J. Dreier, S. Meier, R. Sasse, and B. Schmidt, Tamarin-Prover Manual Security Protocol Analysis in the Symbolic Model, 2019 [Online]. Available: https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf

[13] D. Basin, C. Cremers, J. Dreier, R. Sasse, “Symbolically analyzing security protocols using Tamarin”, *ACM SIGLOG News*, vol. 4, no. 4, 2017, pp. 19–30 [Online]. Available: https://hal.archives-ouvertes.fr/hal-01622110/file/tamarin-tool.pdf

[14] Q. Do, B. Martini, and K. R. Choo, “The role of the adversary model in applied security research”, *Comp. & Secur.*, vol. 81, pp. 156–181, 2019 (DOI: 10.1016/j.cose.2018.12.002).

[15] W. Diffie and M. Hellman, “New directions in cryptography”, *IEEE Trans. on Inform. Theory*, vol. 22, no. 6, pp. 644–654, 1976 (DOI: 10.1109/TIT.1976.1055638).

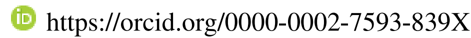
[16] The Illustrated TLS Connection [Online]. Available: https://tls.ulfheim.net/

[17] L. C. Paulson, “Inductive analysis of the Internet protocol TLS”, *ACM Trans. on Inform. and Syst. Secur.*, vol. 2, no. 3, pp. 332–351, 1999 (DOI: 10.1145/322510.322530).

[18] G. Ferro, TCP SYN Cookies – DDoS defence, 2008 [Online]. Available: https://etherealmind.com/tcpsyn-cookies-ddos-defence/

[19] E. Rescorla and N. Modadugu, Datagram Transport Layer Security Version 1.2, 2012 [Online]. Available: https://tools.ietf.org/html/rfc6347

Piotr Remlein received his M.Sc. and Ph.D. degrees from Poznań University of Technology (PUT), Poznań, Poland in 1991 and 2002, respectively. In 2018, he received a D.Sc. degree from PUT. He has been employed at PUT since 1992, currently as an Associate Professor at the Institute of Radiocommunications, Faculty of Computing and Telecommunications. His scientific interests cover wireless networks, communication theory, error control coding, cryptography, digital modulation, continuous phase modulation, mobile communications, and digital circuit design. Dr. Remlein is the author of more than 120 papers, presented at national and international conferences and published in communications journals. He also acts as a reviewer for international and national conference and journal papers. He is a Senior Member of IEEE Communications Society and IEEE Information Theory Society.


https://orcid.org/0000-0002-7593-839X
E-mail: piotr.remlein@put.poznan.pl
Institute of Radiocommunications
Poznań University of Technology
Pl. M. Skłodowskiej-Curie 5
60-965 Poznań, Poland

Urszula Stachowiak received her B.Eng. degree in Information and Communication Technologies from the Faculty of Computing and Telecommunications, Poznań University of Technology, in February 2020. Since March 2020 she has been a master’s student majoring in Computing and specializing in the Internet of Things. From July 2019 to November 2020, she was employed as a Telecommunications Analyst at Comarch, and has been working as a Software Engineer at Intel Corporation since December 2020. Her interests include topics related to the security of IoT and formal proving of communication protocol security.

https://orcid.org/0000-0002-6892-6876
E-mail: urszula.stachowiak@student.put.poznan.pl
Faculty of Computing and Telecommunications
Poznań University of Technology
Pl. M. Skłodowskiej-Curie 5
60-965 Poznań, Poland