DOI: 10.15439/2022F169
ISSN 2300-5963 ACSIS, Vol. 30

Formal verification of security properties of the Lightweight Authentication and Key Exchange Protocol for Federated IoT devices

Michał Jarosz
Cybernetics Faculty
Military University of Technology
Warsaw, Poland
michal.jarosz@wat.edu.pl

Konrad Wrona
NATO Cyber Security Centre /
Military University of Technology
The Hague, Netherlands / Warsaw, Poland
konrad.wrona@[ncia.nato.int,wat.edu.pl]

Zbigniew Zieliński
Cybernetics Faculty
Military University of Technology
Warsaw, Poland
zbigniew.zielinski@wat.edu.pl

***Abstract*—The federated nature of many crucial Internet of Things (IoT) applications introduces several challenges from a security perspective. To address critical challenges related to the authentication and secure communication of IoT devices operating in federated environments, we propose a new authentication and key exchange protocol based on a distributed ledger. Our protocol uses the unique configuration fingerprint of an IoT device and does not require secure storage in participating IoT devices. To validate the correctness of our design, we have performed formal modeling and verification of the security properties, using two different verification tools: Verifpal and the Tamarin prover.**

I. Introduction

THE APPLICATIONS of Internet of Things (IoT) expand rapidly to many mission-critical areas, such as smart health care and Humanitarian Assistance and Disaster Relief (HADR) [1]. Many of these applications are based on the concept of *federation* in which IoT devices operated by different federated partners must communicate with each other securely.

Many protocols are used for authentication and key exchange between devices described in the literature. An overview of such protocols is presented in [2]. The proposed solutions are based on a variety of approaches, including Public Key Infrastructure (PKI) [3], blockchain [4, 5, 6], and shared keys [7, 8]. An advantage of our proposal is its focus on the federated environment, where devices belonging to one organization (and regulated by its security policy) can be accessed and used by other organizations belonging to the federation.

Federating separate IoT administrative domains introduces several challenges from a security perspective. One of the fundamental challenges is establishing an effective identity and access management (IAM) framework, which is necessary to ensure trust and secure communication between federated IoT devices [9]. The particularly critical IAM challenge is the authentication and authorization of federated IoT devices.

To address these challenges, we propose in Section II a Lightweight Authentication and Key Exchange Protocol for Federated IoT (LAKEPFI) that supports flexible authenticated key exchange based on Hyperledger Fabric (HLF) [10]. The solution uses the unique configuration fingerprint of an IoT device and does not require secure storage space in participating devices. Our protocol enables secure communication of heterogeneous federated IoT devices, including devices equipped with additional security hardware, such as Physical Unclonable Functions (PUF) [11], and devices characterized by very limited computing resources such as Arduino.

Constructing a new security protocol is an error-prone process. Therefore, it is essential to verify its security properties using various techniques. The process of verifying the security properties of protocols is a common operation. Any protocol used to communicate with devices must undergo this process. In this article, we will focus on describing the verification of the security properties of the developed protocol. For this purpose, we used two tools: Verifpal and Tamarin. These tools have previously been used for the verification of some commonly used IoT communication protocols [12].

In Section III we briefly introduce formal modeling and verification approaches that can be used to validate the security properties of LAKEPFI. In Sections IV and V, we present and discuss formal modeling and verification of the security properties of LAKEPFI using Verifpal and the Tamarin prover. In particular, our aim is to prove formally that the designed protocol provides: 1) message secrecy; 2) message authentication; 3) freshness.

II. Lightweight authentication and key exchange for federated IoT devices

A. Federated IoT architecture

The main goal of the proposed protocol is to establish efficient and secure communication between devices belonging to different organizations. The main participants in this protocol are IoT devices that belong to other organizations and services on the organization's side: the application gateway and distributed ledger.

Fig. 1 shows the general architecture of the components that are used in the LAKEPFI.

The IoT device consists of two components:

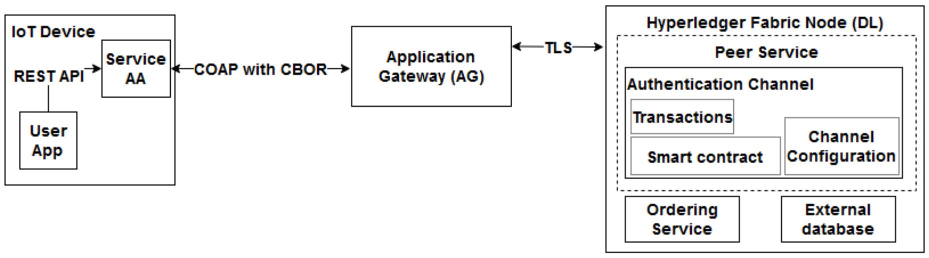

Fig. 1. Component architecture and interconnection scheme.

1) AA Service (Authentication and Authorization Service), responsible for communicating with the application gateway using the solution we describe;
2) User App, the user application that connects to the AA service is agnostic to our framework. Each user application uses a unified interface provided by the AA service.

Our protocol is an application layer protocol, and although it can be used in combination with existing lower-level security protocols to provide increased defense-in-depth, it is agnostic to these lower-layer protocols. In our proof of concept, for efficiency reasons, Constrained Application Protocol (CoAP) [13] with Concise Binary Object Representation (CBOR) [14] is used for communication between the device and the application gateway (AG). Communication between the application gateway and distributed ledger nodes uses Transport Layer Security (TLS). The application gateway is responsible for the following:

1) Conversion from CoAP to TLS and vice versa, due to the fact that Hyperledger Fabric requires TLS with certificates, while IoT devices must use lightweight protocols;
2) Filtering out of invalid and malicious messages;
3) Load balancing, therefore, ensuring that the IoT device does not need to know the whole distributed ledger architecture, which can change.

The last component is the distributed ledger node, which stores all smart contracts and has access to all channels. In our implementation, we used Hyperledger Fabric as a distributed ledger implementation. The Hyperledger Fabric Node consists of two services:

1) *Peer*, responsible for the verification and recording of transactions in the distributed ledger;
2) *Orderer*, responsible for the creation of a block.

B. Description of the protocol

The proposed protocol is based on the use of unique parameters of the IoT device for its authentication. Within this protocol, three main methods are described:

1) Registration of an IoT device in a distributed ledger;
2) Communication of an IoT device with a distributed ledger node;
3) Communication between IoT devices, especially those belonging to different organizations within the federation.

C. Registration phase

The first operation is device registration. This is required for the device to communicate with other devices and services. The registration process can be performed in two different ways: 1) connect directly to the application gateway; or 2) put the device at the destination and secure the communication for the duration of the registration with a one-time login and password. During the registration phase, the device sends values to the application gateway for the parameters that define the device. Each parameter should have an appropriate entropy. This is a kind of fingerprint from the IoT device. Depending on the capabilities of the IoT device, the parameters recorded in the array may include the following:

1) Unique configuration data - in this case, the unique data of the device is used. It can be hardware data, e.g., device serial number, memory card serial number, etc., and it can be software data, e.g., partition IDs, file system IDs, keys stored on the device,
2) PUF data - data from the PUFs embedded in the IoT device,
3) Random strings (for example, keys) - in this case, the device generates random strings with required entropy and needs to store them securely. This method is used only if neither 1) nor 2) can be used.

In the first and second cases, the device does not store the key in its storage; it is enough to hold a program to obtain values of specific parameters. The device performs an appropriate operation to obtain parameters, e.g., a system command to obtain a particular parameter. The second and third cases are prepared for devices with a minimum of 50 kB RAM and 250 kB flash memory, that is, the so-called class C2 [15]. However, in case 2, the device must support PUF.

In the registration process, a set of parameters P_A consisting of its parameters $\{p_1, p_2, ..., p_n\}$ is written in the distributed ledger DL. The communication diagram is shown in Fig. 2. As written in the beginning of this section, there are two options for securely sending P_A from the IoT device A to the application gateway AG. After receiving the array P_A, the application gateway AG generates a new identifier ID_A for the device A. The ID_A along with the parameter array P_A is stored in the

$$\begin{tikzcd}
A \arrow[r, "P_A"] & AG/DL \\
A & AG/DL \arrow[l, "{ID_A, PN, nonce, E_A}"']
\end{tikzcd}$$

Fig. 2. Communication between the IoT device and a ledger node during the registration phase

$$\begin{tikzcd}
A \arrow[r, "{ID_A, PN_{A\to DL}, nonce_{A\to DL}, E_{A\to DL}}"] & AG/DL \\
A & AG/DL \arrow[l, "{ID_A, PN_{DL\to A}, nonce_{DL\to A}, E_{DL\to A}}"']
\end{tikzcd}$$

Fig. 3. Communication between an IoT device and the distributed ledger

distributed ledger DL. After the success of storing these data in the ledger DL, a response is generated to the IoT device A. In the response, the identifier ID_A and the current timestamp t are sent to the device ID_A in encrypted form E_A. The encryption key K is generated by the distributed ledger node DL_i. This key K is the result of a hash function from the concatenation of a subset $\{1, ..., n\}$: $PN_{DL\to A} = \{n_1, n_2, ..., n_k\}$ of the parameters $K = H(p_{n_1}|p_{n_2}|...|p_{n_k})$ sent by the device. The parameters used for encryption are chosen at random. In addition to ciphertext $E_A = E[K; nonce; (ID_A, t)]$, where $E[]$ is the encryption function to encrypt (ID_A, t) using K and $nonce$, AG also sends the identifier in plaintext ID_A , the nonce value $nonce$, and the identifiers of the parameters $PN = \{p_{n_1}, p_{n_2}, ..., p_{n_k}\}$, whose values were used to create the key K, are also sent.

Based on the number of parameters $PN = \{p_{n_1}, p_{n_2}, ..., p_{n_k}\}$ that are sent in the response, the device knows which parameters were used to generate the key K so it can recreate the key K and thus decrypt the message. After decryption, it checks if the timestamp t sent in reply differs more than 5 seconds from the current one. Then it checks if the identifier ID_A sent in plaintext and the ciphertext E are the same. If all these operations were successful, the device is registered and then uses the received identifier ID_A and the generated set of parameters P_A.

D. Communication procedure between IoT device and distributed ledger node

When a device A communicates with an application gateway AG that will perform a task, the device A encrypts the data $data$ it wants to send to the gateway AG in the same way as described for the process of generating a response from the ledger DL during device registration. The communication diagram is shown in Fig. 3. The device A selects a subset of parameters $\{1, ..., n\}$: $PN_{A\to DL} = \{n_1, n_2, ..., n_k\}$ and generates a key $K_{A\to DL} = H(H(p_{n_1})|H(p_{n_2})|...|H(p_{n_k}))$ from them. The key $K_{A\to DL}$ together with the random generated nonce value $nonce_{A\to DL}$ is used to encrypt the data $data$ and the current timestamp $t_{A\to DL}$: $E_{A\to DL} = E[K_{A\to DL}; nonce_{A\to DL}; (data, t_{A\to DL})]$. To the application gateway AG, device AA sends: an identifier ID_A, a nonce $nonce_{A\to DL}$, parameter numbers $PN_{A\to DL}$ that define the parameters used to generate the key $K_{A\to DL}$ and a ciphertext $E_{A\to DL}$. The application gateway AG sends these data to the distributed ledger node DL_i to decrypt the ciphertext $E_{A\to DL}$. The distributed ledger node DL_i is capable of generating a key $K_{A\to DL}$ based on the number of parameters $PN_{A\to DL}$ and the identifier ID_A. Using this key $K_{A\to DL}$ and the nonce value $nonce_{A\to DL}$ it performs decryption of the ciphertext $E_{A\to DL}$. The timestamp $t_{A\to DL}$ stored in the ciphertext $E_{A\to DL}$ and the current one are verified. If the ciphertext $E_{A\to DL}$ was successfully decrypted and the timestamp $t_{A\to DL}$ is correct, the application gateway AG receives the encrypted $data$. Based on $data$, it determines what should be executed and performs the requested operation. The response $response$ is generated in the same way. The device A receives the identifier ID_A, the new nonce value $nonce_{DL\to A}$ used in the response $response$ encryption process, the parameter numbers $PN_{DL\to A}$ that define the parameters used to generate the key $K_{DL\to A}$ and the ciphertext $E_{DL\to A} = E[K_{DL\to A}; nonce_{DL\to A}; (response, t_{DL\to A})]$. The device A after receiving the message is capable of generating the key $K_{DL\to A}$ and thus decrypting the ciphertext $E_{DL\to A}$ and getting the response $response$.

E. Communication procedure between IoT devices

To communicate with each other, it is necessary to authenticate the devices that want to communicate with each other and verify that such communication is authorized. The communication diagram is shown in Fig. 4. The device that wants to establish communication is the device A with ID_A. This device sends the ID_B of the device B to the application gateway AG. The device A can get the ID_B of device B in many ways, e.g., it can receive the ID_B of device B from another device, it can have a record that needs to communicate with that device, or device B can announce that it has a certain type of information. The identifier ID_B is secured in a way that is identical to the data in the device communication process with the distributed ledger DL described in the previous subsection. The application gateway AG sends the request to decrypt to the distributed ledger node DL. The node DL returns the decrypted identifier ID_B. The application gateway AG then sends a request to create a key $K_{A\leftrightarrow B}$ for communication between devices A and B. DL verifies based on the rules stored in the DL authorization channel whether A and B can communicate. If so, then it generates the key $K_{A\leftrightarrow B}$. To create a key $K_{A\leftrightarrow B}$, DL selects a subset of k-elements of $p_{n_1}, p_{n_2}, ..., p_{n_k}$ from the parameter array P_B of the device B, which is stored in the ledger. The key $K_{A\leftrightarrow B}$ is constructed similarly to the previous processes; only in this case is the current timestamp $t_{A\leftrightarrow B}$ is also included: $K_{A\leftrightarrow B} = H(p_{n_1}|p_{n_2}|...|p_{n_k}|t_{A\leftrightarrow B})$. The response to device A must send the key $K_{A\leftrightarrow B}$, timestamp $t_{A\leftrightarrow B}$, and parameter numbers $PN_{A\leftrightarrow B}$ that were created to create the key $K_{A\leftrightarrow B}$. The response is secured in the same way as in the previous

$$A \xrightarrow{ID_A, PN_{A\to DL}, nonce_{A\to DL}, E_{A\to DL}} AG/DL$$

$$A \xleftarrow{ID_A, PN_{DL\to A}, nonce_{DL\to A}, E_{DL\to A}} AG/DL$$

$$A \xrightarrow{ID_A, PN_{A\leftrightarrow B}, nonce_{A\to B}, t_{A\leftrightarrow B}, E_{A\to B}} B$$

$$A \xleftarrow{ID_B, PN_{A\leftrightarrow B}, nonce_{A\leftarrow B}, t_{A\leftrightarrow B}, E_{A\leftarrow B}} B$$

Fig. 4. The procedure of communication between IoT devices

cases. The device A decrypts the message in the same way as in the previous case. If everything is correct, the device A can now send $data$ to the device B. To do this, it has to generate a new nonce $nonce_{A\to B}$. Using the key $K_{A\leftrightarrow B}$ received in the response and the nonce value $nonce_{A\to B}$, it can encrypt $data$ and the timestamp $t_{A\leftrightarrow B}$ that will be sent to the device B: $E_{A\to B} = E[K_{A\leftrightarrow B}; nonce_{A\to B}; (data, t_{A\to B})]$. Then device A sends to device B: ID_A, $PN_{A\leftrightarrow B}$, $t_{A\leftrightarrow B}$,$nonce_{A\to B}$, $E_{A\to B}$.

The device B after receiving the message can decrypt the ciphertext $E_{A\to B}$ because the key $K_{A\leftrightarrow B}$ was created by the distributed ledger node DL based on the parameters P_B of the device B and the timestamp $t_{A\leftrightarrow B}$. Using the key $K_{A\leftrightarrow B}$ and the nonce $nonce_{A\to B}$, the device B decrypts the ciphertext $E_{A\to B}$. If device B successfully decrypts the ciphertext $E_{A\to B}$, this means that the key $K_{A\leftrightarrow B}$ has been created by the distributed ledger node DL and, therefore, has been issued to the authorized entity A. To secure the response, the device B uses the same key $K_{A\leftrightarrow B}$ but with a new nonce value $nonce_{A\leftarrow B}$. Furthermore, the new timestamp $t_{A\leftarrow B}$ is generated in ciphertext. The new ciphertext is built: $E_{A\leftarrow B} = E[K_{A\leftrightarrow B}; nonce_{A\leftarrow B}; (result, t_{A\leftarrow B})]$. Therefore, the answer contains: $ID_B, PN_{A\leftrightarrow B}, nonce_{A\leftarrow B}$, $t_{A\leftrightarrow B}, E_{A\leftarrow B}$.

III. Modeling and verification

Building a secure communication protocol requires verification of its security properties. Such properties include, for example, authentication of the communicating parties, the confidentiality and integrity of the transmitted data, and the equivalence property; that is, an attacker cannot distinguish between any two stages of the protocol.

Formal verification can confirm (or deny) that the protocol is secure. A protocol can be mathematically formalized by [16]:

1) *Symbolic (Dolev-Yao [17]) model*, in which cryptographic primitives are represented as black boxes, messages are terms in these primitives, and the attacker is restricted to using these primitives. This model makes the execution of automatic proofs relatively easy. The symbolic model is robust to attacks if no attack can occur for each trace. By trace, we mean one run of the protocol. Research progress on this type of model is considered highly advanced.
2) *Computational model*, in which messages are strings of bits, cryptographic primitives are functions that operate on these strings, and the attacker is any probabilistic Turing machine. This mode is generally used by cryptographers. A computational model is robust to attacks if no attack can occur for every trace except those with a negligibly small probability. The computational model is more realistic, but still, since it is only a model, it will not give us an answer to whether the developed protocol is resilient to some attacks, such as side-channel or physical attacks. In the case of computational models, most of the time, cryptographic properties have to be proven manually. The proofs in this model are more difficult to perform and analyze. This type of modeling is much less mature. Therefore, we did not use this type of modeling.

It is worth mentioning that, regardless of the modeling approach used, even minor changes to the protocol require a revision of the model used for the analysis. Examples of tools that use the symbolic model include:

1) AVISPA [18] uses a modular and expressive High-Level Protocol Specification Language (HLPSL). It supports many back-end tools to verify a model, and the result of the model verification is easy to interpret. The implemented techniques offer protocol analysis, such as protocol falsification (by finding an attack on the input protocol) and abstraction methods for finite and infinite sessions, among others. AVISPA is currently not widely used in papers.
2) Verifpal [19] uses an intuitive language to describe the model and the result is easy to interpret. Verifpal can validate forward secrecy, key compromise, impersonation, and other advanced queries. It also supports unbounded sessions, fresh and random values, and other valuable features of the symbolic model. However, users cannot define their primitives. It is one of the youngest tools and is still being actively developed.
3) ProVerif [20], like the previous tools, provided predefined and reusable primitives. However, contrary to AVISPA and Verifpal, new primitives can also be defined. The way of modeling in ProVerif is similar to the approach used in Verifpal and AVISPA, but ProVerif uses a different verification method. If ProVerif checks that a security property is fulfilled, then it is indeed so, but ProVerif cannot always prove the properties of the tested protocol.
4) Tamarin prover [21] in its capabilities and popularity is similar to ProVerif; Tamarin prover has a different way of modeling protocols than the other tools. In the case of the Tamarin prover, we model the rules of transition between states in the protocol, while every operation within the protocol is modeled in other protocols. Each rule in Tamarin prover has an input (describes what it takes in), an output (what the result is), and facts. The proof in the case of Tamarin is to verify whether the

lemma we describe is true or not. Similarly to ProVerif, the Tamarin prover is not always able to prove the properties of the tested protocol.

For the analysis of our protocol, we have selected Verifpal and Tamarin. These formal verification tools are described in more detail in the next section. Both Verifpal and Tamarin are widely accepted and are currently used by researchers and practitioners to analyze security protocols. Verifpal supports fast modeling of a protocol and verification of its basic properties (message secrecy and authentication). On the other hand, the Tamarin prover is a more powerful tool that can allow us to verify a protocol in more detail. Both tools also verify the model automatically. Note that these tools work with an unbounded number of sessions, making the problem undecidable [16].

When modeling a security protocol, several issues need to be taken into account [22]:

1) One should assume the correctness of low-level cryptography mechanisms; for example, when using encryption or random value generation, one should assume that these functions work correctly.
2) Appropriate assumptions should be made about external services, e.g., network services such as time servers and trusted third parties.
3) Since protocol messages often need to pass through various middle boxes, such as firewalls and guards, and the protocol participants may be mobile or connected via unreliable communication channels, it has to be assumed that the messages may have different forms or may not reach the other participant.
4) Some protocols require early negotiation to determine the cryptographic primitives to be used, which affect the level of security and performance.
5) The reaction of participants to the occurrence of an error should be appropriately modeled, as proper error handling affects the security of the protocol.

Verifpal offers a more native way of writing the protocol, making it more readable and easier to learn. Verifpal defines two types of an attacker. The first type is *active attacker*, which can intercept all protocol messages, modify them, and inject his messages. The second type is *passive attacker*, which can only intercept messages sent between protocol participants. An attacker is free to use any combination of actions available to him. Verifpal has defined cryptographic primitives such as (a)symmetric encryption, authenticated encryption [23], Diffie-Hellman key exchange, digital signature, and a secure hash function. Verifpal does not allow for the definition of additional cryptographic primitives. The verification result is easy to interpret; precise information about what has been changed and what property is not met. Verifpal is one of the few tools that examines the unlinkability property. Unlinkability is a situation where for two observed values, the adversary cannot distinguish between a protocol execution in which they belong to the same user and a protocol execution in which they belong to two different users [19].

Tamarin prover offers automatic and interactive theorem proving models [21]. Although the Tamarin prover automatically generates proofs, user interaction is sometimes required. The result of the proof itself also brings inconclusiveness to the problem. The analysis is based on labeled multiset rewriting rules to specify protocols and adversary capabilities, a guarded fragment of first-order logic to determine security properties and functions, and equational theories to model the algebraic properties of cryptographic protocols. Furthermore, all events related to security properties are annotated with a time point $t \in Q$, and a basic comparison of time points can be used. Tamarin prover applies a constraint-solving algorithm based on backward search and heuristics that attempts to validate or falsify security properties. This approach requires significant knowledge and experience in formal modeling [24]. Therefore, modeling the protocol using the Tamarin prover is complex, and the result of model verification is also rather difficult to interpret.

IV. Verifpal

Modeling in the Verifpal tool first involves specifying the participants (parties) who will use the protocol. In the model itself, we define what operations a participant will perform and what data are transmitted between them. Finally, there are questions about the security properties of the data transmitted between participants. The role of Verifpal is to confirm the secrecy of transmitted data, confirm its authentication, and verify its freshness. Using Verifpal, three LAKEPFI operations were modeled:

1) Device registration,
2) Communication IoT device with Hyperledger Fabric,
3) Communication IoT device with other IoT device.

In this article, we will only describe in detail the operation of communicating an IoT device with a Hyperledger Fabric node because the other operations are very similar to this one. However, we will discuss model verification results for all three operations at the end of this subsection. During modeling, we followed the rules described in the III section.

First, we configure the mode in which the attacker should operate. In our case, an active attacker can influence the messages sent. Then, we define what the device knows. The device $DeviceA$ knows its ID, which is public, so it also knows HLF (Hyperledger Fabric node) and the attacker. Additionally, the device knows the secret data $dataA$ it wants to send. In the real-life protocol flow, the device generates the parameter numbers $paramnumbersAS$ that are used to create the key. However, in the case of the protocol model, these values are not generated, but are sent and therefore must be included in the message.

The device generates timestamp $timestamp$ and nonce $nonceAS$. In a real-life protocol flow, the device would now generate the $keyAS$ key based on the set of specific parameter values. However, Verifpal does not allow this operation. Therefore, $keyAS$ is generated as a random value. The device concatenates $dataA$ and $timestamp$. The result of this concatenation is then encrypted using $keyAS$ and $nonceAS$. The result of encryption is the ciphertext $EdataAS$.

```
attacker[active]
principal DeviceA[
knows public ID
knows private dataA
generates paramnumbersAS
generates timestamp
generates nonceAS
knows private keyAS
dataToEncryptAS = CONCAT(dataA, timestamp)
EdataAS =
  AEAD_ENC(keyAS, dataToEncryptAS, nonceAS)
]
```

The $DeviceA$ sends to HLF the $paramnumbersAS$, $nonceAS$ and the ciphertext $EdataAS$. This information is learned by the attacker at this point.

```
DeviceA -> HLF: paramnumbersAS, nonceAS, EdataAS
```

HLF also knows the key $keyAS$, which should normally be generated using $paramnumbersAS$ and a set of parameters. Then using $keyAS$ and $nonceAS$ it decrypts $EdataAS$. The result is split, leading to the data $DataAS$ and the timestamp $timestampFromA$. In real implementation, the current timestamp is compared with $timestampFromA$, while in Verifpal, the ASSERT function does not affect anything. HLF generates a response $reply$, which will be sent to $DeviceA$. It also generates a new timestamp $timestampToA$ and $paramnumbersSA$ that, like $paramnumbersAS$, are not used but are sent. The $reply$ is concatenated with $timestampToA$. The result is encrypted using a new key $keySA$ and a new value $nonceSA$. A ciphertext $EdataSA$ is created. As before, the new key is generated as if it were a random value, rather than derived from $paramnumbersSA$ and the parameter array. Of course, the key $keySA$ is marked as private, so it is not known to the attacker.

```
principal HLF[
knows private keyAS
DdataAS = AEAD_DEC(keyAS, EdataAS, nonceAS)
DataAS, timestampFromA = Split(DdataAS)
generates timestampNow
_ = ASSERT(timestampFromA, timestampNow)
generates reply
generates timestampToA
generates paramnumbersSA
dataToEncryptSA = CONCAT(reply, timestampToA)
generates nonceSA
knows private keySA
EdataSA =
  AEAD_ENC(keySA, dataToEncryptSA, nonceSA)
]
```

HLF sends $paramnumbersSA$, $nonceSA$, and $EdataSA$ to $DeviceA$. At this point, the attacker learns these values.

```
HLF -> DeviceA: paramnumbersSA, nonceSA, EdataSA
```

The final step is for $DeviceA$ to decrypt the response. For this, $DeviceA$ knows the key $keySA$, which, as in the previous steps, is not created from $paramnumbersSA$ and the parameter array. The ciphertext $EdataSA$ is decrypted using $keySA$ and $nonceSA$. Finally, the timestamp contained in the ciphertext is compared with the current one.

```
principal DeviceA[
knows private keySA
DdataSA = AEAD_DEC(keySA, EdataSA, nonceSA)
DataSA, timestampFromHLF = Split(DdataSA)
generates timestampNowReply
_ = ASSERT(timestampNowReply, timestampFromHLF)
]
```

The very end of the above listing defines queries concerning the properties that Verifpal has to check. For each operation, there are three properties: confidentiality of transmitted data, authentication of transmitted data, and freshness.

```
queries[
confidentiality? dataToEncryptSA
confidentiality? dataToEncryptAS
authentication? DeviceA -> HLF: EdataAS
authentication? HLF -> DeviceA: EdataSA
freshness? EdataAS
freshness? EdataSA
]
```

When verifying the registration operation and communication of the IoT device with the Hyperledger Fabric node, Verifpal found no errors and therefore confirmed confidentiality, authentication, and freshness.

On the other hand, for the third operation, communication between IoT devices, Verifpal found two errors: failure to authenticate HLF and lack of freshness when sending data from HLF to $DeviceA$. However, both errors are false positives. In both cases, Verifpal sets the value of $nonceSA$ to nil (null), and this situation in a real run will return an error and $DeviceA$ will reject the message. As we mentioned in Section III, modeling does not assume error handling, and here we have a good example.

In Verifpal, we had to apply some simplifications:

1) The key is a generated value, not a value generated from the parameters of device;
2) The parameters themselves are also a generated value and not a list with numbers;
3) There is no way to compare timestamps (although there is an ASSERT function, which is not used in Verifpal).

V. Tamarin prover

The approach to modeling in Tamarin prover is quite different from that in Verifpal. For the Tamarin prover, think of a protocol as a set of states where information not passed to another state is forgotten. The conditions themselves are written as lemmas, which the Tamarin prover verifies. As with Verifpal, Tamarin prover allowed us to confirm secrecy, authenticity, and freshness. In the case of the Tamarin prover, three operations have been modeled:

1) Device registration,
2) Communication IoT device with Hyperledger Fabric,
3) Communication IoT device with other IoT device.

As in Section IV, we will describe in detail the operation of communicating an IoT device with a Hyperledger Fabric node. At the end of this section, we present the results for both operations.

First, we need to define the operations that we will use. The value after / indicates the number of arguments that the operation will take. In addition, two functions are defined: decrypt and verify.

```
theory Device2Ledger
begin
functions:
aead/3, decrypt/2, verify/3, true/0
equations: decrypt(aead(k, p, a), a, k)=p
equations: verify(aead(k, p, a), a, k)=true
```

In the first rule, we generate a key that is shared between two parties. The function Fr means that key is random. Like Verifpal, here there is no possibility of generating the key as described in the actual implementation. In the set of facts, we specify that $Client$ and HLF know our key. We send the associated client $Client$ with the key key and the HLF node HLF with the same key to the next state.

```
rule Setup:
[ Fr(~key) ]
    --[ID_Client($Client,~key),
    ID_Server($HLF,~key)]->
[!Identity($Client, ~key),
    !Identity($HLF, ~key)]
```

The next state takes in a new nonce value $nonce$ that is generated on input to this state, and two bindings, $Client$ and HLF with the key key. In the set of facts, we have written that there is communication between the client and the server using the key key and the value $nonce$. The output is a state in which we send a message from $Client$ to HLF with $nonce$ and the ciphertext $aead(key,' Data', \ nonce)$.

```
rule Client_1:
[ Fr(~nonce), !Identity($Client, key),
!Identity($HLF, key) ]
    --[Client2HLF($Client,$HLF,
    <key,~nonce>)]->
[ Out(<$Client,$HLF,~nonce,aead(key,
    'Dane',~nonce)>)]
```

The next rule defines a state that takes as input the newly generated nonce value $nonce2$, the binding of HLF to key, and the message from the previous state. In the set of facts, we define that there is communication between HLF and $Client$ using key and $nonce2$, and we verify the message. When leaving this state, we send a message from HLF to $Client$ with $nonce2$ and a response $reply$ encrypted with key and $nonce2$.

```
rule HLF_1:
let EncMessage = aead(key,'Dane',nonce)
in
[ Fr(~nonce2),!Identity($HLF, key),
In(<$Client,$HLF,nonce,EncMessage>) ]
    --[ HLF2Client($HLF,$Client,
      <key,~nonce2>),
    Eq(verify(EncMessage,nonce,key),true
    ) ]->
[ Out(<$HLF,$Client,~nonce2,
    aead(key,'Respond',~nonce2)>)]
```

In the last rule, we define that the state receives as input a binding between $Client$ and key, as well as a message from HLF to $Client$. In the set of facts, we verify the message. This state is the final state.

```
rule Client_2:
let EncMessageFromHLF =
        aead(key,'Respond',nonce2)
in
[ Fr(~nonce3),!Identity($Client, key),
In(<$HLF,$Client,nonce2,EncMessageFromHLF>
)]
    --[Eq(verify(EncMessageFromHLF,nonce2,
    key),true)]->
[ ]
```

In addition, we have defined some restrictions. The first two restrictions specify that $Client$ cannot communicate with $Client$ and HLF cannot communicate with HLF. In DenyClient2Client, we define that when $Client$ sends a message to HLF, $Client$ cannot create a message and send it to itself. Similarly, in the DenyServer2Server constraint, we specify that a HLF node cannot send a message to itself.

```
restriction DenyClient2Client:
"
All Client HLF key nonce #i. (
  Client2HLF(Client,HLF,<key,nonce>) @ #i
    & not(Client = HLF)
) ==> not (Ex #j nonce2 .
Client2HLF(Client,Client,<key,nonce2>) @j)
"
restriction DenyServer2Server:
"
All Client HLF key nonce #i. (
  HLF2Client(HLF,Client,<key,nonce>) @ #i
    & not(Client = HLF)
) ==> not (Ex #j nonce2 .
HLF2Client(HLF,HLF,<key,nonce2>) @j)
"
```

For the dual_clients restriction, we specify that there do not exist two $Clients$ with the same key.

```
restriction dual_clients:
"
All Client key #i. (
    ID_Client(Client,key) @ #i
) ==> not (Ex #j Client2 .
ID_Client(Client2,key) @j))
"
```

The first lemma defines the secrecy of the key. For each $Client$, HLF, Key, two $nonces$ and two moments i and j in the situation: when there is a connection of $Client$ to HLF at moment i and HLF to $Client$ at moment j and moment j is after i and $Client$ cannot also be HLF then there is no such moment k that there is a revealed Key at moment k.

```
lemma Key_Secrecy:
"
All Client HLF Key nonce nonce2 #i #j. (
Client2HLF(Client,HLF,<Key,nonce>) @ #i &
HLF2Client(HLF,Client,<Key,nonce2>) @ #j &
#i < #j &
not (Client = HLF)
)==> not(Ex #k1 . K(Key) @ #k1)
"
```

In the second lemma, we check the freshness. For each client $Client$ and node HLF and key t and two moments i and j in the situation: when there is a connection from $Client$ to HLF at moment i and HLF to $Client$ at moment j and moment j is after i where there are:

1) There is no such user $Client2$ at moment $i1$ that there is communication from $Client2$ to HLF with the same key $KeyCH$ at moment $i1$ when i is equal to $i1$;
2) There is no user $HLF2$ at moment $j1$ that there is communication from $HLF2$ to $Client$ with the same key $KeyHC$ at moment $j1$ when j is equal to $j1$.

```
lemma freshness:
"
All Client HLF KeyCH KeyHC #i #j. (
Client2HLF(Client ,HLF,KeyCH) @ #i &
HLF2Client(HLF, Client ,KeyHC) @ #j &
#i < #j &
not(Client = HLF)
)==> (not (Ex Client2 #i1 .
Client2HLF(Client2 ,HLF,KeyCH)
  @i1 & not (#i1 = #i)) & not(Ex HLF2 #j1 .
HLF2Client(HLF2, Client ,KeyHC)
  @j1 & not (#j1 = #j)))
"
```

For $Client$ authentication, you need to prove that $Client$ has a key that is used to protect the data before sending them. To do this, we have defined that for any $Client$, HLF, $nonce$, and $nonce2$, at moments i and j, there is a communication between $Client$ and HLF where HLF sends a response to the request of $Client$ and there is always a moment k such that $Client$ has the key that was used to secure the communication before starting that communication.

```
lemma auth_Client:
"
All Client HLF key nonce nonce2  #i #j . (
Client2HLF(Client ,HLF,<key,nonce>) @ #i &
HLF2Client(HLF, Client ,<key,nonce2>) @ #j &
#i < #j &
not(Client = HLF)
)==> Ex #l . ID_Client(Client ,key)
    @l & #l < #i
"
```

The same is true for HLF authentication; it must also have key before communication can begin where key is used.

```
lemma auth_HLF:
"
All Client HLF key nonce nonce2  #i #j . (
Client2HLF(Client ,HLF,<key,nonce>) @ #i &
HLF2Client(HLF, Client ,<key,nonce2>) @ #j &
#i < #j &
not(Client = HLF)
)==> Ex #k . ID_Server(HLF,key)@k & #k < #i
"
```

In the Tamarin prover, we used the same simplifications as in Verifpal:

1) The key is a generated value, not a value generated from the parameters of devic;
2) The parameters themselves are also a generated value and not a list with numbers;
3) There is no way to compare timestamps.

As with Verifpal, Tamarin prover also confirmed key secrecy, freshness, and authentication of the $Client$ and the HLF node during communication of the IoT device with the Hyperledger Fabric node. We also verified secrecy, authentication, and message freshness for the other operations. In both cases, the result was positive. Both tools enforced identical constraints on us. Using the two tools allowed us to confirm that the protocol we developed is secure and provides the required security features. Using two tools reduced the probability that our proposed protocol did not meet our criteria because both tools use different deduction mechanisms.

VI. Conclusions and future work

We have performed a formal modeling of the LAKEPFI protocol and presented the results of the analysis of its security properties. For our modeling and analysis, we have used two complementary formal verification tools, Verifpal and Tamarin. By choosing tools that use significantly different protocol models, we tried to minimize the possibility of erroneous verification. Based on the models developed using these tools, we verified the security properties of the protocol, such as message secrecy, authentication, and freshness. The choice of tools used to verify the protocol was not straightforward. We did not find a tool that was able to completely model our protocol. After testing many tools, we chose two that were closest to meeting our requirements. The difficulty in modeling our protocol lies mainly in generating the key to authenticate communication.

In the case of both tools, successful modeling of LAKEPFI required introducing some specific assumptions and simplifications. For example, the key is a randomly generated value; not a value generated explicitly from the device parameters. We have previously checked the randomness property of the created keys and, therefore, consider this assumption valid. Another limitation is that it is impossible to generate an array and randomly select the values from which the key should be created. Therefore, the array indexes are sent explicitly, which may affect the verification results. The final limitation is the inability to compare timestamps. In Verifpal, the possibility of evaluating values of timestamps is limited to the (unused) ASSERT function, while in Tamarin, one can define which events follow each other in time. However, one cannot compare the variables that store timestamps. The result of the time comparison influences the result of the execution of the protocol. If their difference is too significant, the message is considered invalid. Moreover, both tools that we used verify symbolic models. A symbolic model abstracts away the details of cryptographic operations and does not consider all implementation details, which could be seen as its limitation.

As part of future work, we plan to verify selected security properties of the LAKEPFI protocol in the computational model and validate the correctness and security of our implementation of the protocol. We also plan to test the performance of the protocol by experimenting with different use cases and configurations of the federated IoT environment.

Acknowledgment

This work has been partially funded by the NATO Allied Command Transformation Innovation Programme of Work and by the SEMACITI project, sponsored by the Ministry of Defense of Republic of Poland as part of the Kosciuszko Programme.

References

[1] N. Suri et al. "Exploring Smart City IoT for Disaster Recovery Operations". In: *Internet of Things (WF-IoT), 2018 IEEE 4th World Forum on*. IEEE. 2018, pp. 463–468.

[2] P. K. Panda and S. Chattopadhyay. "A secure mutual authentication protocol for IoT environment". In: *Journal of Reliable Intelligent Environments* 6.2 (June 2020), pp. 79–94.

[3] Z. Qikun et al. "Multidomain security authentication for the Internet of things". In: *Concurrency and Computation: Practice and Experience* ().

[4] U. Khalid et al. "A decentralized lightweight blockchain-based authentication mechanism for IoT systems". In: *Cluster Computing* 23.3 (2020).

[5] G. Shaoyong et al. "Master-slave chain based trusted cross-domain authentication mechanism in IoT". In: *J of Network and Computer Applications* 172 (2020).

[6] C. Chen et al. "A secure blockchain-based group key agreement protocol for IoT". In: *The Journal of Supercomputing* (Feb. 2021).

[7] M. Santos et al. "FLAT: Federated lightweight authentication for the Internet of Things". In: *Ad Hoc Networks* 107 (2020).

[8] M. Alshahrani and I. Traore. "Secure mutual authentication and automated access control for IoT smart home using cumulative Keyed-hash chain". In: *J of Inf Sec and App* 45 (2019), pp. 156–175.

[9] N. Kshetri. "Can Blockchain Strengthen the Internet of Things?" In: *IT Professional* 19.4 (2017), pp. 68–72.

[10] N. Haur et al. *Building decentralized applications with Hyperledger Fabric and Composer*. Packt Publishing, 2018.

[11] A. Babaei and G. Schiele. "Physical Unclonable Functions in the Internet of Things: State of the Art and Open Challenges". In: *Sensors* 19.14 (2019).

[12] K. Hofer-Schmitz and B. Stojanović. "Towards formal verification of IoT protocols: A Review". In: *Computer Networks* 174 (2020), p. 107233.

[13] C. B. Z. Shelby K. Hartke. *The Constrained Application Protocol (CoAP)*. Request for Comments (RFC) 7252. IETF, 2014.

[14] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. IETF, Oct. 2013.

[15] C. Bormann, M. Ersue, and A. Keranen. *Terminology for Constrained-Node Networks*. Request for Comments (RFC) 7228. IETF, May 2014.

[16] B. Blanchet. "Security Protocol Verification: Symbolic and Computational Models". In: *Principles of Security and Trust*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–29.

[17] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208.

[18] A. Armando et al. "The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications". In: *Computer Aided Verification*. Ed. by K. Etessami and S. K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 281–285. ISBN: 978-3-540-31686-2.

[19] N. Kobeissi. *Verifpal User Manual*. Manual. Symbolic Software, 2021. URL: https://verifpal.com/res/pdf/manual.pdf.

[20] B. Blanchet et al. *ProVerif 2.04: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. 2021.

[21] D. Basin et al. "Symbolically Analyzing Security Protocols Using Tamarin". In: *ACM SIGLOG News* 4.4 (2017).

[22] M. Abadi. *Security Protocols and their Properties*. 2001.

[23] H. Krawczyk. "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)" In: *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 310–331.

[24] A. Zanatta. *Comparison of Tools for the Verification of Cryptographic Protocols*. 2021. URL: https://github.com/AlessandroZanatta/Verification-of-Cryptographic-Protocols.