Formal Verification of Object Security for Constrained RESTful Environments (OSCORE)

Bachelor Thesis
Project ID: 2722500

Written by Mikkel Kaj Andersen at the **IT-University of Copenhagen**
under supervision by Alessandro Bruni

May 15, 2019

Contents

Abstract

Object Security for Constrained RESTful Environments (OSCORE) is a security protocol built for IoT. As the nodes and networks in IoT are limited in hardware it is crucial that OSCORE is both lightweight and secure. Since OSCORE is a candidate for being proposed standard under the IETF, a formal verification is mandatory. The protocol claims that it ensures Confidentiality, Integrity, Request-response binding and Non-replayability. This paper intends to formally verify that OSCORE indeed adheres to these properties.

This thesis conducts a formal verification of OSCORE by creating a model of the protocol using a tool called ProVerif. ProVerif is a cryptographic protocol modelling language used for the automatic verification of protocols. We proved that confidentiality holds, and that integrity and request-response binding holds under non-injective terms, since it was not possible to derive a result for the properties under injective conditions. A result for non-replayability could not be derived in any case, but after tweaking the model we were able to confirm that shared security contexts must be synchronised in order to hinder replay attacks.

Acknowledgements

I am grateful for Alessandro Bruni's counselling and help throughout the project. His help got me through many issues that I encountered during the development of my thesis. Thanks for always keeping the door open.

I would also like to give a special thanks to the members of the IETF CoRE group Francesca Palombini, Göran Selander, John Mattsson, & Karl Norrman for the time and interest devoted to the project.

1 Introduction

Given the growing threat from cyberattacks powered by the Internet of Things (IoT)[1], it seems clear that new protocol standards for IoT should be proposed, such that lightweight and secure communication can be achieved end-to-end on the application layer even in the presence of proxies.

One protocol being proposed is Object Security for Constrained RESTful Environments (OSCORE) [10]. In order for OSCORE to be proposed standard, a formal verification of its given properties should be conducted.

This bachelor project formally verifies that the protocol, OSCORE, is safe to implement in the coming generation of IoT devices. The verification of OSCORE is based on the automatic cryptographic protocol verifier ProVerif, used to confirm that the protocol complies with the security properties: *Confidentiality*, *Integrity*, *Request-response binding*, and *Non-replayability*.

ProVerif is a modelling language used to create a *model* of the protocol. Based upon the model, the protocol can be queried to validate if it is possible to violate the desired security properties. If such an violation exists, ProVerif outputs the sequence of actions to be taken by the attacker in order to execute the attack. The model is an abstract interpretation of the protocol for which certain aspects are simplified. Functions, like an encryption function, are treated as a black-box, and are in this example expected to output a ciphertext based on a plaintext. The details of how this is done is deemed irrelevant for the analysis. This is advantageous in the sense that a protocol should not rely on, f.x. whether SHA-256 or SHA-512 has been used for hashing, but rather we should focus on the fact that hashing is used, and how that impacts the protocol. This is known as the symbolic model, which differs from the computational model in which f.x knowing most of a decryption key will allow an attacker to guess the rest. For OSCORE it was necessary to abstract the enumeration of requests and responses away, while replay-protection and atomic operations were discarded due to the constraints of ProVerif. Workarounds and other tools were explored but due to time constraints proper results could not be achieved. Furthermore, it was not possible to model the Observe option or the security context negotiation protocol due to time constraints. They are therefore out of scope.

We dive deeper into ProVerif and its usage in Chapter 3, and the specific implementation of the model in Chapter 4. The next chapter is intended to give the reader an introduction to OSCORE.

[1] https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/

2 Object Security for Constrained RESTful Environments (OSCORE)

Object Security for Constrained RESTful Environments (OSCORE) is a protocol currently in development by the Internet Engineering Task Force (IETF). At the time of writing, the protocol is still in draft and should not be considered anything other than 'work-in-progress'. In this chapter OSCORE is explained, along with some of the other standards proposed by the IETF, which OSCORE uses.

2.1 Motivation

OSCORE is a lightweight security protocol intended for IoT devices and constrained environments. The rationale for designing OSCORE is to provide secure transmissions end-to-end between two nodes, while supporting proxy operations. This works as an alternative to DTLS which only secures transmissions hop-by-hop[2]. The underlying architecture builds on CoAP and CBOR, which are explained later in this chapter.

OSCORE encrypts the message payload of the CoAP request or response, via a symmetric key which both parties derive through a shared security context. The establishment of the security context may be done through a key establishment protocol like the Diffie-Hellman key exchange or EDHOC [9].

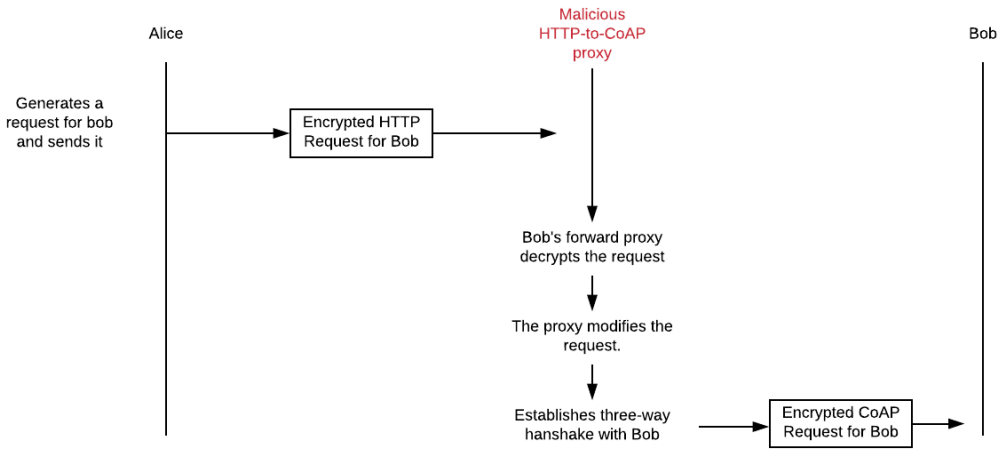

Figure 1: A malicious proxy can undetectably modify requests and responses. The request that Alice sent could be changed from a POST to a DELETE and would subsequently result in content stored at Bob being removed.

Datagram Transport Layer Security (DTLS) is a protocol that provides security for communicating applications by preventing tampering and eavesdropping. It is used by CoAP to encrypt messages between endpoints. As CoAP supports proxy operations and caching, this encryption must be ended at the proxy, because the message carries information intended for the proxy that specifies how it should be treated. Therefore the proxy may tamper with the message, and afterwards forward it to the receiving endpoint which will not detect that the message has been corrupted. It is naïve to believe that every intermediary in the communication-chain can be trusted, and even if they can, such a construction is fragile in the sense that an attacker may take possession of a node involved in the

[2]This term is used when a message is processed in an intermediary and not just forwarded to the next node

network. The idea with OSCORE is to hide as much of the message as possible while still supporting proxy operations. This means that a genuine proxy can still perform its intended functionality, and a malicious proxy is strongly hampered in its ability to tamper with the message.

2.2 The protocol

OSCORE is a symmetric encryption scheme for CoAP. It protects the requested resource, the message payload, the request method and certain options through encryption. As certain options should be readable by the proxy, they are divided into classes. A class represents how an option should be protected. An option in the CoAP message may belong to one of the three following classes:

- Class E (Encrypted): Added to the payload of the message and then encrypted. Visible only to the intended recipient.
- Class I (Integrity Protected): Added as a regular option to the message, as well as being integrity protected through its use in the encryption and decryption of the message payload. Visible to all.
- Class U (Unencrypted): Added as a regular option to the message and is completely unprotected. Visible to all.

OSCORE relies on an established security context between two endpoints, which contains a shared secret key, and unique identifiers for each of the two endpoints. It may also contain a shared salt for extra randomness. Together this forms the security context (Example in table 1), which may also contain optional parameters.

	Alice	Bob
Common Context	Master Secret: 'P$55VV03D' ID Context: 'BobAndAliceID'	Master Secret: 'P$55VV03D' ID Context: 'BobAndAliceID'
Sender Context	Sender ID: 'Alice' Sender key: '9bj4ngXJ9fD' Sender Sequence Number: 2	Sender ID: 'Bob' Sender key: 'xJcxv39kgpo' Sender Sequence Number: 3
Recipient Context	Recipient ID: 'Bob' Recipient key: 'xJcxv39kgpo' Replay window: [0, 1, 2]	Sender ID: 'Alice' Sender key: '9bj4ngXJ9fD' Replay window: [0, 1]

Table 1: Example of the security context used in OSCORE. For simplicity sake, the data is not represented as byte strings.

From the identifiers and the shared secret a 'sender key' is derived for each endpoint and stored in a 'sender context', used for sending messages. It also stores a Sender Sequence Number (SSN), used for enumerating requests. The sender context is mirrored in the 'recipient context', which holds the same key and ID now used for receiving and verifying messages. The recipient context also stores a replay window used for replay protection. One of the reasons two keys are used is that two messages should not be encrypted with the same nonce and key, therefore having two keys allows the same nonce to be reused, and an endpoint only has to keep track of the nonces it itself has generated.

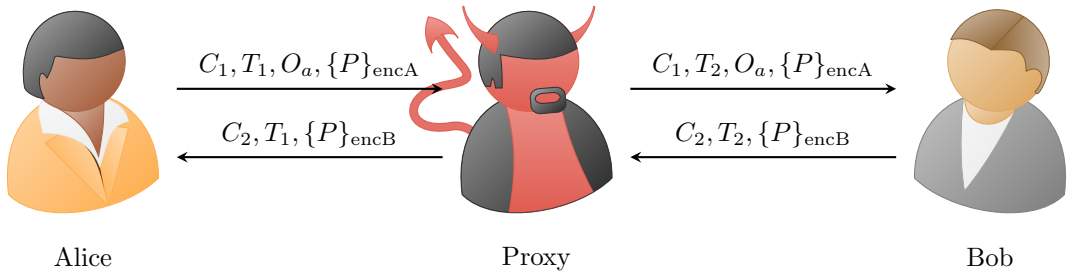

Figure 2: Example of Alice sending a request for Bob through a proxy

In the remainder of this chapter we illustrate OSCORE by example. In this case Alice would like to communicate to Bob using the context of Table 1. An illustration can be found in Figure 2. Alice begins with identifying the security context shared with Bob. Assuming Alice has an unencrypted CoAP message then the request code, Class I & E options and the payload, must be assembled and encrypted using both her sender key '$9bj4ngXJ9fD$' and SSN '2', among others. The ciphertext output from the encryption will now be used as the new message payload. As the plaintext used in the encryption is now protected, the request code should be removed and replaced with the dummy code C_1. The Class E options are also removed from the message, and it is subsequently sent. In order for Bob (who might be a server running multiple security contexts with different endpoints) to derive the same key, he must identify the security context shared with Alice. Therefore, the sender ID (known as recipient ID for Bob) is included in O_a, along with the SSN used in the encryption and, optionally, an ID Context, which is used to help Bob identify a recipient context.

Value	Definition
C_1	A placeholder request/response code.
C_2	A placeholder request/response code.
T_1	A token which uniquely identifies the request.
O_n	The OSCORE option, indicating OSCORE processing. Contains the Sender ID, SSN, and ID Context.
P_{encA}	The encrypted payload of the request.
P_{encB}	The encrypted payload of the response.

Table 2: Values and their definition in Figure 2.

Bob will then proceed to identify a security context that matches the recipient ID received in O_a and derive the recipient key '$9bj4ngXJ9fD$'. Bob asserts that the SSN received have not been used in the replay window already. In this case it has not, he therefore decrypts and accepts the request, for which he generates a response protected with his sender key, '$xJcxv39kgpo$'. The response will not have any identifiers, as Alice is expected to match T_1 of the request, to the response, and identify the security context that way. Alice then verifies the response and the session is finished. A detailed version of the exchange can be found in Appendix 6.1. We finish this chapter by summarising which other standards OSCORE uses in order to ensure its security properties.

The request is protected using CBOR Object Signing and Encryption (COSE) [8] which in turn is used with an Authenticated Encryption With Additional Data (AEAD) [6] algorithm. COSE is used for bringing signing, encryption, and other security services to the CBOR (Explained in Chapter 2.4) data format. The default COSE AEAD algorithm used in OSCORE is AES-CCM-16-64-128 which is used in conjunction with [8, Section 5.3] to encrypt and decrypt.

AEAD is defined in [6] as: "[...] a form of encryption that, in addition to providing confidentiality for the plaintext that is encrypted, provides a way to check its integrity and authenticity. [...] AEAD adds the ability to check the integrity and authenticity of some Associated Data (AD), also called "additional authenticated data", that is not encrypted". Whenever additional authenticated data (AAD) is mentioned it is a reference to this description.

For completeness, a step-by-step approach of how Alice performs OSCORE to secure their communications can be found below. They are also summarised in [10, Section 8], where the step-by-step approach of Bob can be found. Given that Alice already has a CoAP message she wishes to encrypt, she must:

1. Compose the AAD to be used.
    - A CBOR array that contains the SSN, Sender ID and Class I options.
2. Compose the plaintext.
    - A bytestring that contains the payload of the CoAP message, Class E options and the request/response code.
3. Compute a AEAD Nonce to be used with the AEAD algorithm.
    - Computed from the Sender ID, Common IV, and SSN. Deriviation of Common IV and Sender Key is described in [10, Section 3.2.1]. Common IV is only used as additional input to the AEAD Nonce.
4. Construct the COSE object as in [8, Section 5.2].
    - Created as a CBOR array that contains both the SSN and the Sender ID (refered to as headers), in addition to the ciphertext payload. The ciphertext is the plaintext encrypted using the Sender Key, AAD, and AEAD Nonce.
5. Replace the original request code with POST and replace the original payload with the ciphertext of the COSE object. The OSCORE option is added which contains the headers of the COSE object. The message can now be sent.

When Bob receives the request he should be able to compose the same data. Since the OSCORE option contains the Sender ID, a recipient context can be identified, and thereby a recipient key. In order to protect against replay attacks, Bob must assert that the SSN received has never been seen before. When Bob is ready to send a response, he uses the AEAD nonce from the request, and his Sender Context to encrypt the plaintext.

In order to avoid replay of responses, Alice will only accept a single response to a request, which means that the token of a response is only accepted once. Since the response must mirror the token used in the request, a modification of the token will result in Alice not finding the security context used.

OSCORE relies heavily on the use of CBOR and CoAP. Therefore we would like to shortly summarise their key features.

2.3 CoAP

The Constrained Application Protocol (CoAP) [11] is a web transfer protocol made for constrained nodes and networks, making it suitable to use with OSCORE. The message format of CoAP may be found in [11, Figure 7], however in order to understand the model of OSCORE we shortly summarise the most important parts of the CoAP message:

- The token, which uniquely identifies a request
- The payload, which will typically include the resource representation from a successful request or response.
- Options, which is a list of zero or more options, where an option is additional information related to the request or response. For example, both the URI path of a request and the payload media type is carried in options.

CoAP is originally designed for UDP, and therefore messages will inevitably arrive out of order, or appear duplicated. This is handled by a lightweight reliability mechanism, which is why the message seen in [11] contains multiple other values. OSCORE does not protect the underlying message layer of CoAP or the token, and therefore the mechanisms in place for UDP (or if you use CoAP over TCP) are not in scope, only the payload, code, and options are.

Finally, as explained in Chapter 2.1, CoAP specifies binding to DTLS for security.

2.4 CBOR

Concise Binary Object Representation (CBOR) is a standardized format often used by applications in order to transport data in binary format from one endpoint to another. It is an alternative to the more popular variant, JavaScript Object Notation (JSON). JSON prioritises readability, which induces more than necessary overhead in its encoding. Therefore, using a binary data format which is faster to encode/decode like CBOR is often recommended when endpoints have limited computing power and memory.

CBOR expands on the key-value pair data format that JSON uses, by allowing a value without a key, and binary values. For example, [17, 20] is a legitimate value and will be encoded to "0x821114", as explained below:

```
82    # Denotes start of an array - the next 2 values are in the array
   11 # unsigned(17)  First value of the array
   14 # unsigned(20) Second value of the array
```

The CBOR data format is used in numerous places throughout the OSCORE protocol. It is most importantly used when encoding the AAD and encrypting/decrypting the plaintext/ciphertext.

In the scenario that an endpoint wishes to encrypt a plaintext with additional authenticated data, then the AAD must follow the authenticated data structure (COSE_Encrypt0 structure) specified in COSE [8, Section 5.2]. A valid authenticated data structure with its Partial IV and kid header set could look like the following:

```
   Enc_structure = CBOR array of
   [
       "Encrypt0"
        h'',
        h'8501810A40412540'
    ]
```

- Which will encode to 0x'8368456E6372797074304048850181 0A40412540' where the first 11 bytes are fixed, as this is what "Encrypt0" and h" will encode to. Line 5 is an example AAD encoded as a bytestring[3].
- It's vital that both endpoint agree upon the data structure, otherwise the receiving endpoint will parse the values incorrectly since there are no keys to indicate the contents of the value. The data structure is only understood through its order of elements.
- Once the structure is in place, the data is now ready to be used in an AEAD algorithm as AAD.

[3]We recommend the use of http://cbor.me/ for playing around with CBOR

2.5 Properties

OSCORE should comply with the following properties [10, Appendix D]:

- **Confidentiality**
  - *An attacker should not be able to determine the plaintext contents of a given OSCORE message or determine that different plaintexts are related*
- **Integrity**
  - *An attacker should not be able to craft a new OSCORE message with protected message fields different from an existing OSCORE message which will be accepted by the receiver.*
- **Request-response binding**
  - *An attacker should not be able to make a client match a response to the wrong request.*
- **Non-replayability**
  - *An attacker should not be able to cause the receiver to accept a message which it has previously received and accepted.*

In order to verify that OSCORE adheres to the claimed security properties, the protocol will be modelled in a tool called ProVerif. The next chapter is intended to give the reader an introduction to how the tool works.

3 ProVerif

In this chapter we give a brief tour of the grammar used in ProVerif, which is necessary to understand the model of OSCORE, in addition to giving an overview of how ProVerif works internally.

ProVerif is a cryptographic protocol modelling language used in the symbolic analysis of protocols. It has been in development since the start 2000s, and it is therefore quite reliable in its ability to reason about the security properties of stateless protocols.

The symbolic analysis of a protocol means that the implementation of cryptographic functions are abstracted away, and instead cryptography is assumed to be perfect. This allows the modeller to focus on the matter at hand: the sequence of events taken by the protocol in order to obtain some security objective. ProVerif takes as input a model written in applied pi-calculus and translates it into a set of inference rules called Horn-clauses. If ProVerif derives that a queried fact is reachable, an attack trace can be obtained which is the sequence of events needed in order to trap the protocol into an attack state which violates the claimed security properties. When ProVerif derives a fact to be reachable it attempts to reconstruct a corresponding attack trace in the applied pi calculus model, which is not always possible due to the over-approximations done by ProVerif. This leads to the fact that there might be an attack after translating the model into Horn clauses which does not correspond to an attack in the original model. There is always the risk that due to the high level of abstraction the protocol might not be safe in a more realistic environment, however if this is an issue, one should consider building the protocol around the *computational model*, which is more realistic, but also more difficult due to the many technical details. Furthermore, ProVerif ships with a built-in Dolev-Yao attacker, which is that the adversary has control of the network. This means they have the capabilities to intercept, replay, transform, insert and delete messages. In the model, they may create and use any value or function whose type is public, in the same sense that they may maliciously pick and choose data from eavesdropped communications in order to violate the model, however they may not break the crypto.

We will now present some of the key features of ProVerif used in our modelling of OSCORE.

3.1 Keywords and functionality

Types & Values

The syntax for declaring a new value is [`x : t`], which will bind the type 't' to the name 'x'. Multiple declarations can be made with the notation [x_1`, ..` x_n `: t`] meaning $x_i \in t$. The availability of a value must always be declared by prepending a value with a scope declaration that determines how restricted the variable is. If the variable is being declared globally (known to all) the variable is prepended with `free`. A value (or function) can be made hidden from the attacker but still known to the processes by appending `[private]`. If the value is created in the local scope of a single process, it must be prepended with `new`. Furthermore, there are two built-in types `bitstring` and

```
1 free secret : bitstring [private].
2
```

Figure 3: Example of declaring a global value only known to the processes and not the attacker

`channel`. Custom types can be declared by [`type x.`] which are useful when a function must restrict its parameters to a certain type, and the `bitstring` type is too generic.

Functions

Functions in ProVerif are similar to declaring a function in an interface of an object oriented language like C# or Java, where the specific implementation is abstracted away. A function is declared by the word `fun`, a name, parameters, and return type. This could for example be an encryption function: [`fun encrypt(bitstring, key) : bitstring.`] which given a bitstring and a custom type 'key' will return a new bitstring that is unintelligible from the original bitstring. This is by default irreversible but it is possible to declare a 'destructor' through the `reduc` keyword which can reverse the operation. It is worth noting that a function used with the same parameters will create the same value.

```
1 reduc forall message: bitstring, k: key; decrypt(encrypt(m, k), k) = m.
2
```

Figure 4: How to reverse the encrypt function. The destructor decrypt is declared. It takes as input a ciphertext and a key. If the ciphertext was encrypted with the same key that is given as input, the plaintext is returned.

Channels

Channels are the main way for processes to communicate. Messages may both be received [`in(c, value : bitstring)`] and sent [`out(c, value)`] on any given channel, here called 'c'. Channels are characterised by being in the ownership of the attacker, and as a result, any data put on the channel will be known to the attacker who can then decide what the intended recipient should receive. It is possible to declare a channel that the attacker does not have ownership of by using the `[private]` notation, and as a result any data sent will not be known to the attacker.

Phases, Queries & Attacker

Each process in the model is running under a phase, where the phase is $\in \mathbb{N}$ initialized at 0. By default, all processes run under phase 0, however, a new phase can be specified by `(phase 1; out(c, k))`, which means that the value k will only be sent out under phase 1. This kind of functionality is useful for validating forward secrecy. Forward secrecy means that in the event that the keys used in a session are disclosed to an attacker, the secrecy of the messages exchanged before the disclosure is still kept. An example of a protocol that ensures forward secrecy even in the case that one party's key is disclosed can be found in Appendix 6.2[4]. A host 'A' uses a key k received in a message encrypted with his public key if he can verify that the message was signed by host 'B' and that the message was intended for him (line 22).

```
1 let (=pkA, =pk(b), k:key) = checksign(y, sk(pk(b))) in
```

When run, the model will attempt to prove that the state where the attacker has 's' is unreachable:

```
1 query attacker(s) phase 0;
2   attacker(s) phase 1;
3   attacker(s) phase 2.
```

As the example protocol is intended to ensure forward secrecy, even in the case that the private key of host B is disclosed (phase 1), the secrecy of the messages exhanged with the key k is kept. Forward secrecy cannot be verified if the private key of host A is disclosed (phase 2) as the attacker will then have access to the key exchanged. As seen above, the model is queried for a state where the attacker

[4]The example is largely based on [1, Figure 3.1.5]

knows the ciphertext encrypted with k called s. It is also possible to query for the execution of an `event`. Events are a way to notify that a certain event has occurred, but does not affect the model otherwise. They may take in parameters, which is useful for modelling a relationsship between the correspondence of two events. As in the following example, the query checks that when the event finishA occurs with a public key, there have been a past event finishB with the same public key.

```
query aPublicKey: pkey; event(finishA(aPublicKey)) ==> event(finishB(aPublicKey)).
```

Tables

A collection of data in ProVerif can be modelled using a `table`. A table is a collection of records, which allows data to be stored across multiple runs of a protocol. Data can be inserted and retrieved by using the `insert` and `get` keywords. As an example, you might have a table that maps a host to its key:

```
table host_key(host, key).
```

It might also be necessary for the fetched record to satisfy some requirement. This is done with the `suchthat` keyword. As an example, a key may be verified only if it and its corresponding owner can be found in a table. Note that tables in ProVerif are private and not accessible to the attacker, unless a process is specifically created to read and output the contents of a table.

```
get host_key(h : host, k : key) suchthat k = BobsKey && h = Bob
```

Tables can be encoded similarly with the usage of private channels [1, Section 6.3.2], and it is therefore assumed that they are similair in their construct. Furthermore, [1, Section 6.3.4] specifies private channels as a source of incompletion due to the complex abstractions that must be made, which can lead to queries that can not be proved, and therefore tables may be risky to use.

When put together, one should be able to reason about the constructed model. We therefore explain the output generated by ProVerif in the next chapter.

3.2 The ProVerif output

There are 3 possible results when running ProVerif on a model:

- The result of the query is true.
  - This means that ProVerif was not capable of reaching a state where the attacker has a value, or an event is executed.
- The result of the query could not be proved.
  - ProVerif could neither achieve a trace that violates the query, and it could not prove the result to be true either. As explained above, this may be due to the usage of tables, or a flaw in the model. ProVerif still outputs a trace which in some cases can be used to conclude a result.
- The result of the query is false.
  - A state was reached which violated the query and a correct trace has been established.

In the case that the query is false, ProVerif will output a trace in order to achieve the attack state. The output can be divided into three parts. Firstly, ProVerif will output its internal representation of each process, where each action done in the process is numbered for reference.

```
(
    {3}in(c, x_32: bitstring);
    {4}let y: bitstring = adec(x_32,sk(pka)) in
    {5}let (=pka,=pk(b),k_33: key) = checksign(y,sk(pk(b))) in
    {6}out(c, enc(s,k_33));
    {7}event finishA(pka)
)
```

Figure 5: Internal representation of process 'A'

Secondly, ProVerif has derived a series of steps that leads to the attack. Each step represents an action taken by the processes which will have a reference to the number explained above. ProVerif attempts to establish a trace by approximating on all possible outcomes, which means that replicas of data will eventually be made and therefore values used in the output are appended a number that uniquely identifies which process they were spawned in. An example of the trace that violates the forward secrecy of Appendix 6.2 because the attacker has the secret key of A is seen in Figure 6.

```
in(c, pk(a)) at {8}

new k_34: key creating k_704 at {9}

out(c, ~M_728) with ~M_728 = aenc(sign((pk(a),pk(b),k_704),sk(pk(b))),pk(a)) at {11}

in(c, ~M_728) with ~M_728 = aenc(sign((pk(a),pk(b),k_704),sk(pk(b))),pk(a)) at {3}

out(c, ~M_740) with ~M_740 = enc(s,k_704) at {6}

out(c, ~M_742) with ~M_742 = sk(pk(b)) at {13}

out(c, ~M_744) with ~M_744 = sk(pk(a)) at {15}
```

Figure 6: Executing the attack. The attacker may decrypt {11} by knowing the key obtained at {15}

Finally, in addition to the attack, ProVerif outputs a proof that the attack is correct. As mentioned earlier, ProVerif works by Horn-clauses, which is either a fact of the form $=> z$, or a piece of conditional logic of the form $v_1..v_i => z$, where z is the new data obtained without any prerequisites or if and only if $v_1..v_i$ is known respectively. The sequence of derivations (or the proof of the attack) outputted by ProVerif may be used in combination with the trace of the attack to verify that the origin of the information used is valid. The output is more descriptive, however, the underlying formulae follows the form of the above.

```
3. By 2, the attacker may know a[].
Using the function pk the attacker may obtain pk(a[]).
attacker(pk(a[])).
```

Figure 7: Example of the clause that allows the attacker to obtain a public key.

The clause seen in Figure 7 may also be obtained by having ProVerif output each clause generated during the solving process. This has been done for Appendix 6.2 and a snippet can be found in Appendix 6.3. In this case, the pk clause can be found on line 2: `attacker(v_44) -> attacker(pk(v_44))`. The value v_44 is used to obtain the respective public key. Inspecting the output should give a good

impression of the attackers capabilities. It showcases that ProVerif ships with a quite sophisticated attacker, which is an advantage to the modeller because he needs to worry less about programming his capabilities.

We will now introduce a practical approach to how ProVerif is best used for modelling.

3.3 Modelling

ProVerif seems best for protocols where the messages exchanged does not rely on state that will change over time. Tables might seem like they can be used to circumvent that issue, but in my experience they can not. As an example, given a table that holds all valid keys for all hosts, no keys can be removed from the table, and a key will therefore always be valid. We therefore consider related tools which circumvents this issue in the next chapter.

The developer of the model should assure that the model of the protocol is correct. One thing he or she can do is to confirm that all events in a process are reachable by using a `event` of arity 0, called `here`. This event should be inserted chronologically at the end of all processes and queried for, in order to assure the developer that the termination of all processes are reachable.

It should also be verified that if the results of the queries outputs an attack, that attack is a flaw in the protocol and not in the model. False attacks are best avoided by an iterative process where the modeller runs ProVerif after every change, and asserts everything performs as expected. In this regard, queries can be used to test the model, where an example of a test could be using the `here` event. Another approach is to make the protocol open to a specific attack that the protocol attempts to safe guard against, and then take steps according to the protocol to remove the attack. This will ensure that the developer remembers to program the attacker with the appropriate functionality in addition to improving the general quality of the model. Furthermore, as the model increases in size, inspecting the output of the attack trace becomes cumbersome and difficult. Time can generally be saved by having ProVerif output the attack trace as a graph[5], which can be used to get a quick overview of the trace instead of inspecting the derivations right away. Once the developer has reasoned about the outputted graph, time can be spent inspecting the proof of the attack.

One should be cautious when making claims based on queries that return true, as there is no trace to verify why the result is unreachable. One can reason about the trace outputted by an attack, but the only argument to a query returning true is that the model is correct. Therefore special consideration should be put into asserting that the result of the query is reachable under other conditions, like if a part of the protocol fails and the keys used are disclosed to an attacker.

The next chapter will discuss how the grammar presented is used to create and query the model of OSCORE.

[5] $proverif -graph ./ *.pv

4 Modelling OSCORE in ProVerif

In this chapter the model of OSCORE in ProVerif will be explained in details. I also cover how the security properties were queried and their results, in addition to how certain parts of the protocol were simplified.

4.1 The protocol

OSCORE was covered in Chapter 2, however, only the process of encrypting the request was explained in depth. There are three additional operations that remains to be covered which are *verifying the request*, *protecting the response* and *verifying the response*. The actions performed are quite similar, so I refer to [10, Chapter 8] for details. The specific differences compared to the encryption of the request are summarised as follows:

- Verifying the request.
  - The receiver must use the Sender ID carried in the request to identify a recipient context. They must also assert that the SSN carried in the request has not been seen before. The AEAD nonce, AAD and recipient key can now be derived in the same fashion as was used for the encryption, which means the recipient can decrypt the payload, and if this succeeds, the replay window is updated with the received SSN.
- Sending the response
  - The response is encrypted by following the same steps as for the encryption of requests, however it is up to the responder if the nonce from the request is used or a new one is generated. If a new nonce is generated the SSN is included in the response and increased by one.
- Verifying the response
  - The endpoint must assert that the token received in the response has not been previously accepted (recall that a token uniquely identifies a request) and matched with a request. In addition, they must either derive a new nonce or use the one from the request.

4.2 Building the model

The model consists of the four phases outlined in Chapter 4.1, which we dive deeper into in Chapter 4.3. In this chapter we explain the fundamentals of the model.

The messages exchanged follows the structure of a modified version of the CoAP message found in [11, Section 3, Figure 7]:

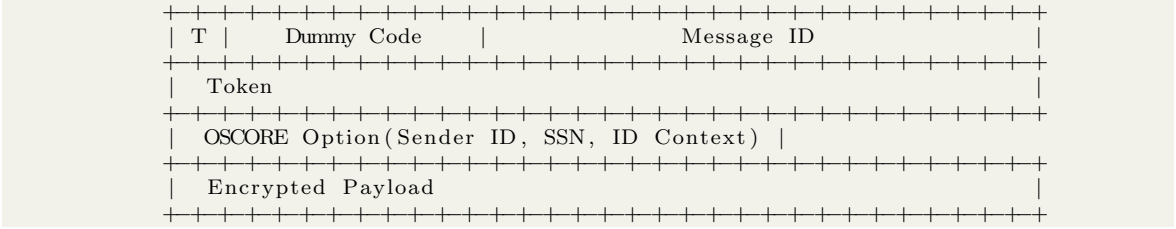

Figure 8: Message model in ProVerif. The full CoAP message would only induce more overhead, so certain parts of the message were discarded.

The message carries only one option, which is the OSCORE option. Additional options could have been added, however it is assumed that it would only introduce more overhead with the same results. That is, adding another Class E or Class I option would just result in an additional parameter for the encrytion and/or AAD. In general it can be concluded that modification or spoofing of options generally results in denial-of-service attacks [10, Appendix D]. If privacy analysis was in scope, this could have been included, however due to time restrictions it was not a consideration. Furthermore, in regards to requests and responses, since a response should be generated from the request, a `bounded_by(bitstring)` function is included which based on a request payload generates a response payload. Additionally, the encrypted message payload for a request or response has been modified to contain a unique identifier obtained from calling `msg1(host, host)` or `msg2(host, host)` respectively.

```
(*Request*)
let plaintext = ((CoAP_GETCode, isLightBulbTurnedOn), msg1(identity, endpoint)) in
(*Response*)
let plaintext = ((CoAP_CONTENTCode, bounded_by(isLightBulbTurnedOn)), msg2(identity,
    endpoint))
```

Figure 9: Example plaintext for a request and response

AEAD

The default AEAD algorithm used by OSCORE is AES-CCM [2] which combines the AES [14, Section 20.3] encryption function with CCM [13]. This type of encryption and counterpart decryption is modelled in Figure 10. The encryption takes a plaintext, nonce, key, and AAD as input to create a ciphertext. If an encrypted message is obtained it can be decrypted with the same AAD, nonce and key used in the encryption. It should not be mandatory for the attacker to verify the integrity of the AAD, therefore there are two ways to decrypt: with, and without the AAD.

```
(*Encryption func*)
fun aeadEncrypt(key, nonce, bitstring, bitstring): bitstring.
(*Decryption func using AAD*)
reduc forall plaintext: bitstring, n: nonce, k: key, aad: bitstring; aeadDecrypt(
    aeadEncrypt(k, n, plaintext, aad), k, n, aad) = plaintext.
(*Decryption func without AAD*)
reduc forall plaintext: bitstring, n: nonce, k: key, aad: bitstring; decrypt(
    aeadEncrypt(k, n, plaintext, aad), k, n) = plaintext.
```

Figure 10: Encryption/Decryption in OSCORE model

HKDF

OSCORE relies on a HKDF (HMAC-based Extract-and-Expand Key Derivation Function) [3] in order to derive the sender key, recipient key, and common IV. The HKDF function is modelled as a function that returns a key given the shared secret and a master salt along with the info object.

```
fun HKDF(bitstring, bitstring, bitstring): key.
```

Figure 11: The HKDF Key Deriviation

The info object is constructed as specified in [10, Section 3.2.1]. This is interpreted into the model as a tuple where it may be used to construct a sender key and common IV using the following construct:

```
(*Common IV*)
let cvInfo = (emptyId, idcontext, AES_CCM, label_iv, alg_key_length(AES_CCM,
    label_iv)) in let CommonIv = HKDF(msecret, msalt, cvInfo) in
(*Sender key*)
let senderInfo = (senderID, idcontext, AES_CCM, label_key, alg_key_length(AES_CCM,
     label_key)) in let senderKey = HKDF(msecret, msalt, senderInfo) in
```

Security context & Request response binding

In order for a host to communicate with another host a security context must be established. The security context for two processes are modelled with a `table` called `security_context_lookup`. The creation of the security context is done through a process called `init_keys`, which receives two hosts from a channel and creates a security context between them. The security context consists of the two sender IDs, a shared secret and salt, and the ID Context. As required in [10, Chapter 3.3], the sender ID and ID Context will always be unique, as they will be freshly generated every time. Ideally, the SSN should be included in the security context, however since it was not possible to model the SSN as an increasing number, a unique `bitstring` called `partial_iv` is created and used as SSN everytime a process sends a request. Furthermore, the association between the token used in the request and the security context used to protect said request is modelled in a similar manner as above. We do not

```
let security_context = (endpoint, senderID, recipientID, idcontext, partial_iv,
    aead_nonce) in
insert token_to_message_lookup(identity, token, security_context);

```

Figure 12: Example of storing the security context, SSN and nonce of a request with a token

store the entire security context with a token, but rather the identifiers, which can be used to look-up a security context. Some simplifications also had to be made to the replay window of the security context, as explained below.

Simplifications

As ProVerif is not optimised for the handling of stateful protocols, it was not possible to include the replay window as part of the security context. Since tables cannot change the state of already inserted items, the replay window was modelled as a table itself:

```
table replay_window(host, bitstring).
```

Figure 13: Replay window in OSCORE

The table contains the host identifier, and all SSNs seen for that particular host. If a message is received and the SSN can be found in the table it's a replay attack, and the process should terminate. This works because all SSN are by definition (using the `new` keyword) unique in the model.

```
get replay_window(=identity, partial : bitstring) suchthat partial = partial_iv in 0
    else
let (plaintextreceived:bitstring, identifier:bitstring) = aeadDecrypt(ciphertext,
  recipientKey, aead_nonce, aad) in
insert replay_window(identity, partial_iv);
```

Figure 14: Updating the replay window

If the SSN has not been received before, then if the request payload can be decrypted, the replay window is updated with the received SSN (Snippet in Figure 14). Note that `0` denotes the termination of the process. Tokens that have been used to accept a response are modelled in a similair manner.

4.3 Running the model

With the fundamentals in place, it can be said that the model acts as outlined in Chapter 2 when asked to reach the event where a response is accepted. The sequence of actions taken were derived by using the approach of inserting the `here` event at the end of the response host as explained in Chapter 3.3. This trace was used to build Figure 15, for which we will now explain the actions performed.

1. The attacker begins with creating the identities of two hosts 'initiator' i & 'responder' r. He sends the identities to the process that will create a security context between them.

2. The process 'init_keys' has created a security context between i & r. The attacker knows the unique identifiers, but only i & r knows the shared secret and salt.

3. The attacker tells i to send a request to r. Now r will perform the following operations:

   (a) They create a new message ID m_{id1}, token T, and SSN.

   (b) They lookup a security context with r using the aforementioned `security_context_lookup`, and if one can be found, a sender key and common IV is derived from it.

   (c) They compose the AAD. It contains the encrypt0 prefix, version, COSE algorithm (AES-CCM), and, more importantly, Sender ID and SSN used to protect the request.

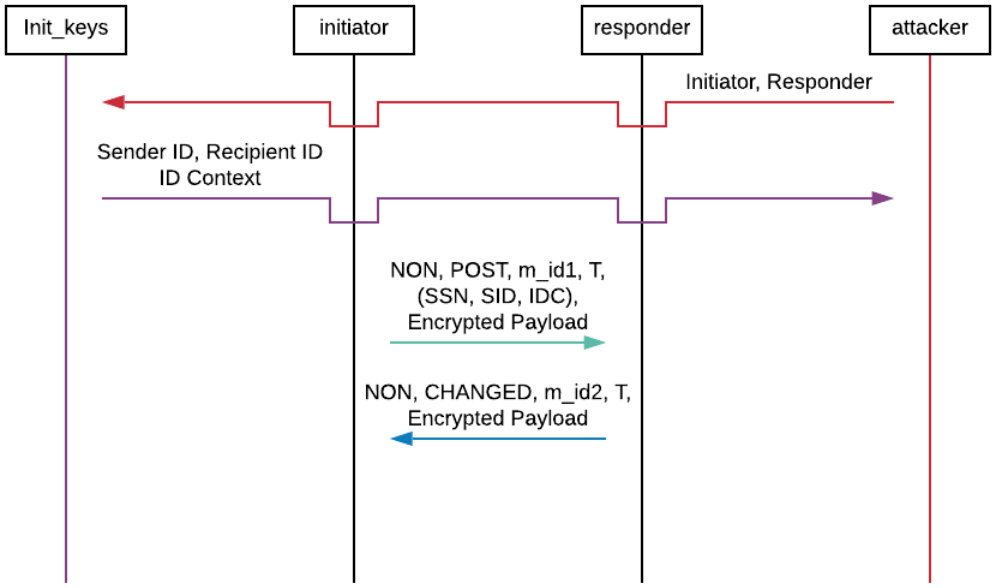

Figure 15: Message exchange in the model

(d) They compose a nonce from the function `aeadNonce`, which given a SSN, Sender ID and Common IV returns a nonce.

(e) They compose the plaintext from the function `msg1(...)`, the request method, and payload. It will now be encrypted using the sender key, AAD, and the nonce.

(f) The OSCORE option is composed. It contains the SSN, Sender ID, and ID Context.

(g) Before sending the message they create an association from T to the security context, nonce, and SSN used, through the `token_to_message_lookup` table.

4. With the message now sent, r will perform the following operations when received:

(a) They repeat step 3b, 3c, and 3d using the Recipient ID, SSN, and ID Context received in the OSCORE option.

(b) They assert that the SSN does not exist in the table `replay_window`, and if found must terminate.

(c) Using the nonce, key and AAD composed in step 4a they attempt to decrypt the encrypted payload. If this succeeds, the SSN is inserted into the `replay_window`.

(d) Based upon the payload of the request and `msg2(...)` a response is generated. The response is encrypted using the Sender Key of r, and the AAD and nonce used to decrypt the request.

(e) A new message ID m_{id2} is created, and the message is sent.

5. When i receives the response, they must:

(a) Verify that T has not been used to accept a response before by checking if the token is stored in the table `used_tokens`.

(b) Based on T identify the security context, SSN, and nonce used in the request/response.

(c) Repeat step 3c, and derive the recipient key.

(d) If the security context identified via the token, and the nonce/AAD of the request can be used to decrypt the response, the table `used_tokens` must be updated with the used token.

The code of the project and the full graph outputted by ProVerif may be found on GitHub[6]. From talks with the authors it was agreed that the security context negotiation protocol specified in [10, Appendix B.2] should be included in the model as well. The protocol was used when two hosts had lost a part of their security context, like the replay window, and had to establish a new one using the existing security context. The final model does not have the protocol included because it was not possible to have the model created reach termination. The verification of the protocol is left as future work, however a ProVerif file of the protocol which does not terminate is included in the GitHub repo.

[6] https://github.com/joakiti/OSCOREProverif

4.4 Verifying the properties

According to Chapter 2.5, OSCORE should adhere to the following properties: *Confidentiality*, *Integrity*, *Request-response binding*, and *Non-replayability*. In this chapter we will present the results of the queries based on the model. The results are discussed in Chapter 4.5.

4.4.1 Integrity, Request-response binding & Non-replayability

Firstly, I will argue that Request-response binding, Integrity, and Non-replayability can all be modelled with the property of *Injective agreement*:

- **Injective agreement** [5, Section 2.4]
  - *Whenever an honest endpoint A receives and accepts an encrypted OSCORE message believing it to be from the honest endpoint B, then B has sent the message believing it was to A, and the endpoints agree on the messages exchanged*

.

Injective agreement can be modelled in ProVerif using an injective correspondence between events. Injectivity means that for each end event e_{end} there have been a distinct occurence of the start event e_{start}. The initiator and responder each has a start and end event which are given a set of parameters. The start event, `beginInitiator(host, host, bitstring)`, is given the identity of the two hosts communicating and the plaintext of the request. The end event, `endResponder`, is given the same set of parameters. Once a request has been encrypted and sent `beginInitiator` is executed, and vice versa for `endResponder`. A similar query is made for sending and protecting the response in addition to receiving and verifying it. The queries may be seen in Figure 16.

```
1 query init:host, res:host, payload: bitstring; inj-event(endResponder(init, res,
      payload)) ==> inj-event(beginInitiator(init, res, payload)) || event (dishonest(
      init)) || event (dishonest(res)).
2 query init:host, res:host, payload: bitstring; inj-event(endInitiator(init, res,
      payload)) ==> inj-event(beginResponder(init, res, payload)) || event (dishonest(
      init)) || event (dishonest(res)).
3
```

Figure 16: Injective agreement in the OSCORE model

The properties are preserved since:

1. Both parties must agree on the content exchanged (Integrity).
2. The same response or request may not be accepted more than once (Non-replayability).
3. A response sent should be the same one received (Token matching).
   (a) We consider an attack on the property as given a possibility to make a response match the wrong request, one would abuse it to have the accepted message contents be different from the original response.

Due to the over-approximations that ProVerif makes, it was not possible to derive a trace that proves injective-agreement, however, we were able to prove *non-injective agreement*.

Non-injective agreement is a weaker property than injective agreement in the sense that the correspondence between events does not have to be a one-to-one match. Under these conditions non-replayability cannot be proved. However, we claim that request-response binding and integrity are proven true in the model but under non-injective conditions.

```
query init:host, res:host, payload: bitstring; event(endResponder(init, res, payload)
    ) ==> event(beginInitiator(init, res, payload)) || event (dishonest(init)) ||
    event (dishonest(res)).
query init:host, res:host, payload: bitstring; event(endInitiator(init, res, payload))
     ==> event(beginResponder(init, res, payload)) || event (dishonest(init)) || event
     (dishonest(res)).

```

Figure 17: Non-injective agreement in the OSCORE model.

4.4.2 Confidentiality

Confidentiality is modelled by checking that the attacker cannot obtain the payload of the exchanged messages between two honest hosts. This is done using the `attacker` query (Explained in Chapter 3.1). Specifically, we ask the attacker to either obtain `msg1` or `msg2` where the event dishonest has not occured.

```
    (*Confidentiality*)
query i:host, r:host; attacker(msg1(i,r)) phase 0 ==> event (dishonest(i)) || event (
    dishonest(r)).
query i:host, r:host; attacker(msg2(i,r)) phase 0 ==> event (dishonest(i)) || event (
    dishonest(r)).

```

Figure 18: Confidentiality between honest hosts i & r

Under the circumstances the property of confidentiality holds in the model. We claimed in Chapter 3 that one should take caution with drawing conclusions on queries which returns true. Therefore a query was created which leaks the master secret used in the security context, and subsequently checks if confidentiality still holds.

```
(*Forward secrecy*)
query i:host, r:host; attacker(msg1(i,r)) phase 1 ==> event (dishonest(i)) || event (
    dishonest(r)).
query i:host, r:host; attacker(msg2(i,r)) phase 1 ==> event (dishonest(i)) || event (
    dishonest(r)).

```

Figure 19: Forward secrecy between honest hosts i & r

As expected, confidentiality of all former messages are sacrificed if the attacker obtains the master key between the two hosts.

4.5 The results

To summarise our findings, confidentiality holds in the model. Looking at the definition for confidentiality, we must assume that it is meant that an attacker can not determine if different ciphertexts are related. It is our assumption, based on the symbolic model, that the attacker can not determine if different ciphertexts are related as a new nonce is generated and used for every ciphertext. This aspect of the property would have to be considered in the computational model to have the highest degree of certainty.

Both integrity and request-response binding holds under non-injective conditions. We believe that these results would be the same under injective conditions, and therefore conclude that the properties hold. As non-injectivity contradicts the purpose of replay protection, we conclude that non-replayability could not be proved under any circumstances.

In order to have the injective agreement query reach a result, certain adaptions were made to the model. The specific cause of why a trace cannot be obtained is still unknown, but it is assumed it has something to do with the use of `tables`.

We consider the mechanism for injectivity in ProVerif to be reliable in many cases, but as our results show, it will sometimes fail when there is a mismatch between the Applied-pi calculus model and the derived Horn clauses. The challenge with Horn clauses is that a fact may reachable any number of times, even if the model does not allow such behaviour. We expect that this is why global state is so difficult to achieve in protocol modelling. Nonetheless, they are therefore not perfect for automatic protocol solvers, but they are still very reliable and their results can easily be reasoned about by humans. In this regard, they strike a good balance between ease-of-use and being realistic enough that we can gain confidence in the protocol being modelled.

4.5.1 Edits to the model

By moving the functionality of `init_keys` into the main process of ProVerif, it was possible to derive a result from the first injective agreement query: `query inj-event(endResponder(...)) ==> inj-event(beginInitiator(...))`. The result of the query returns false, that is, an attack exists. However, the model derives an attack that should not be possible with the replay protection mechanisms used. The trace obtained finds that there is not a one-to-one correspondence between `beginInitiator` and `endResponder`, meaning that it's possible to replay the same request twice. This is because processes in ProVerif can be paused during their execution. This is problematic when running the process under replication as the verification of the request and the update of the replay window should occur atomically. Instead a process can pause execution after verifying the request, without having updated the replay window. The consequence of this is that a replicated process will verify and accept the same request, until the replay window is updated. This is why the replay attack occurs.

In Figure 20 the replay can be seen. The scenario of the attack is described in [10] as the issue of sharing a security context between two servers that are not synchronised. With this, we have proved that it is an issue if a security context is not synchronised, however the authors have already made three possible solutions for synchronising, which are defined in 'Group OSCORE' [12]. They are out of scope of this document.

The consequence of moving the functionality of `init_keys` into the main process was that the model becomes restricted by only having a single security context between two honest hosts, and therefore the attacker is limited in his ability to actively participate in the model. This does pose some limitations on the possible attacks. For example, in 1995 the Needham-Schroeder protocol was found

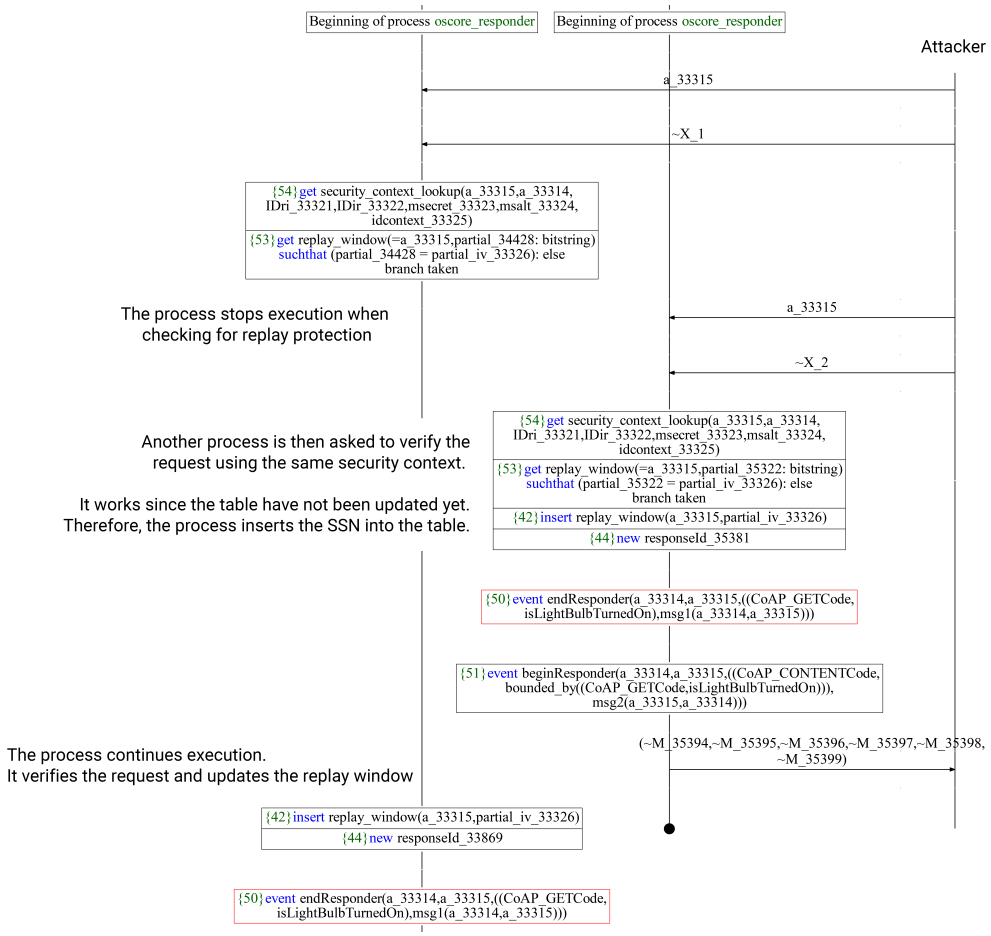

Figure 20: ProVerif's replay attack. For illustration purposes, the other processes have been edited away. The full figure of the graph can be found on the GitHub project. Note that '$\tilde{X}\_1$' & '$\tilde{X}\_2$' are the same message.

vulnerable to a man-in-the-middle attack, which required that the intruder was an active host in the protocol [4]. In another attempt, the model was created where each host had an additional security context with the attacker, but no additional attacks could be found.

An alternate solution to solve the second injective agreement query, `inj-event(endInitiator(...) ==> inj-event(beginResponder(...).` was also found. It suffers from the same issues that the replay window does. It is therefore possible to replay responses. The attack is exactly the same, just from response to request. It will therefore not be discussed further. The graph of the trace may be found in the project on GitHub, under the name 'replayResponseAttack.pdf'.

4.6 Other tools

ProVerif is a useful tool for exploring design flaws in protocols, however, it has still been challenging to use in a stateful protocol. As discussed in Chapter 4.5, the model suffers from the lack of atomic operations and insufficient state properties. A limitation found in the ProVerif modelling language. The necessary functionality for modelling that a value may belong to different sets over time, and that a single horn clause can derive multiple facts does not exist in ProVerif. Had it been possible, the clause for updating the replay window could have looked like this (in abstract terms): `attacker(notSeen(SSN, req)) => updateReplayWindow(SSN), VerifyRequest(req)`.

Security researchers have presented other tools which attempts to circumvent the issue of global states in automated analysis of security protocols, with promising results. We therefore recommend that the future verification of OSCORE builds the model in a suitable framework like StatVerif or GSVerif. They are frameworks that can be used in the verification of stateful protocols. They run on top of ProVerif which means that the existing model can be used. Furthermore, the project repository already contains a draft of OSCORE in another tool called AIF-Omega, an extension to AIF, developed by Alessandro Bruni and Sebastian Mödersheim, published in [7]. AIF-Omega was a promising tool, but due to time restrictions it had to be abandoned, we therefore leave the verification of injective-agreement as future work.

5 Conclusion

In this work, a formal analysis of the OSCORE protocol was conducted using ProVerif. We did not identify any issues in OSCORE, and we can therefore based on the model recommend its future use. However, the model was not capable of proving all properties, due to the over-approximations done by ProVerif.

The general state of the model works as expected. It follows the sequence of operations proposed by OSCORE to send and verify requests and responses. This was validated by having the model generate a trace for its termination, where we saw that there was a strong correspondence between the model and the documentation. We therefore believe our results to be sound. It has been proven that confidentiality holds, and that integrity and request-response binding holds under non-injective terms. A result for non-replayability could not be derived under any circumstances. We therefore made some tweaks to the model, which made the queries intended to prove non-replayability output a result. The model outputted a false attack, from which we concluded that shared security contexts must be synchronised in order to hinder replay attacks.

An attempt was made to rewrite the model into AIF-omega, but no presentable results could be achieved. We therefore leave the unproven properties as well as the further development of the project files as future work.

References

[1] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. Proverif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial, 2018.

[2] Russ Housley. Using AES-CCM and AES-GCM Authenticated Encryption in the Cryptographic Message Syntax (CMS). RFC 5084, November 2007. https://rfc-editor.org/rfc/rfc5084.txt.

[3] Dr. Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010. https://rfc-editor.org/rfc/rfc5869.txt.

[4] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer, 1996. https://www.cs.utexas.edu/ shmat/courses/cs6431/lowe.pdf.

[5] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings 10th Computer Security Foundations Workshop*, pages 31–43. IEEE, 1997. https://pdfs.semanticscholar.org/db04/4d703941fefd5b4609034a3730aebf6ebcd1.pdf.

[6] David McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, January 2008. https://rfc-editor.org/rfc/rfc5116.txt.

[7] Sebastian Mödersheim and Alessandro Bruni. Aif-omega: Set-based protocol abstraction with countable families. In *International Conference on Principles of Security and Trust*, pages 233–253. Springer, 2016. https://www.imm.dtu.dk/ samo/aifom.html.

[8] Jim Schaad. CBOR Object Signing and Encryption (COSE). RFC 8152, July 2017. https://rfc-editor.org/rfc/rfc8152.txt.

[9] Göran Selander, John Mattsson, and Francesca Palombini. Ephemeral Diffie-Hellman Over COSE (EDHOC). Internet-Draft draft-selander-ace-cose-ecdhe-13, Internet Engineering Task Force, March 2019. https://datatracker.ietf.org/doc/html/draft-selander-ace-cose-ecdhe-13.

[10] Göran Selander, John Mattsson, Francesca Palombini, and Ludwig Seitz. Object Security for Constrained RESTful Environments (OSCORE). Internet-draft, Internet Engineering Task Force, March 2019. https://datatracker.ietf.org/doc/html/draft-ietf-core-object-security-16.

[11] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014. https://rfc-editor.org/rfc/rfc7252.txt.

[12] Marco Tiloca, Göran Selander, Francesca Palombini, and Jiye Park. Group OSCORE - Secure Group Communication for CoAP. Internet-Draft draft-ietf-core-oscore-groupcomm-04, Internet Engineering Task Force, March 2019. https://datatracker.ietf.org/doc/html/draft-ietf-core-oscore-groupcomm-04.

[13] Doug Whiting, Russ Housley, and Niels Ferguson. Counter with CBC-MAC (CCM). RFC 3610, September 2003. https://rfc-editor.org/rfc/rfc3610.txt.

[14] Lawrie Brown William Stallings. *Computer Security, Principles and Practice*. Pearson, global edition edition, 2018.

6 Appendix

6.1 Operations performed by the endpoints

The figure seen below is an illustration of how OSCORE secures transmissions end-to-end between hosts. The figure should be read from left to right, where we go through what operations are performed on an example request and subsequent response. The steps taken are quite similar for the response, so there is no example response message in the figure. Note that the figure does not take replay protection into consideration.

CoAP Request ==
- CoAP Version
- Confirmable/ non-confirmable
- Length of Token
- Request Code(type)
- Message ID

} Header

- Token value
- Options
- Payload

} Body

CoAP to OSCORE proxy mapping

1. Retrieve Sender Context (Using sender ID) for target resource/recipient

2. Generate Additional Authenticated Data → A CBOR array which contains

The OSCORE Version: (Only 1 right now)

Array of algorithms for the AEAD. (Only 1 right now)

Sender ID

Partial IV

Options that should be integrity protected

3. Generate plaintext → Plaintext is equivalent to a CoAP message without header.

- Request Code(type)
- Options
- Payload

- Consisting of: the 'Code' of the original CoAP message

-All Inner Option message fields of the original CoAP message(class E)

-The payload of the original CoAP message.

4. "Encode" the partial IV → The Partial IV is sender sequence number

Afterwards, increase sender sequence number by one.

Compute the AEAD nonce from Sender ID, Common IV, and Partial IV

5. Encrypt the COSE Object (Plaintext, AAD) with Sender Key → The COSE object contains the following fields:
Protected: Empty
Unprotected:
[Partial IV,
'kid'(sender id) parameter,
'kid context'
]

Plaintext object:
- The Code of the orginial CoAP message (not dummy code)
- Inner option message fields (Class E)
- Payload of the original CoAP message.

Ciphertext: computed from secret key, AEAD nonce, plaintext, and AAD.
Decrypted from the same

The COSE object must be compressed after encryption.

The sender should put class E options in the Ciphertext

The sender should put 'class I' options in the AAD

'Class I and Class U options' are transported in the header or in options

6. Format the CoAP message to OSCORE by adding the OSCORE message option

7. Send the message!

Sent message:

Code: POST OR FETCH (Just a dummy code) - 0.02

Token: Random hex - 0xFF

OSCORE: [Partial IV, kid context, kid parameter]
If-Match-Outer: 1
Uri-Host Outer: 1
...
Payload: Encrypt({Plaintext}, Secret Key)
The nonce and AAD is also used as input

1. Receive the message

2. Discard all class E options (If a option that should be encrypted is visible in the header, we instead discard it)

Code: POST OR FETCH (Just a dummy code) - 0.02

Token: Random hex - 0xFF

OSCORE: [Partial IV, kid context, kid parameter]
REMOVED
Uri-Host Outer: 1
...
Payload: Encrypt({Plaintext}, Secret Key)

3. Decompress the COSE Object.
- Use the Kid parameter to receive a recipient context.
- Check Partial IV is unique

Code: POST OR FETCH (Just a dummy code) - 0.02

Token: Random hex - 0xFF

OSCORE: [**Partial IV**, kid context, **kid parameter**]
REMOVED
Uri-Host Outer: 1
...
Payload: Encrypt({Plaintext}, Secret Key)

4. Compose the AAD

5. Compute the AEAD Nonce

AEAD Nonce is computed from Recipient ID, Common IV, and Partial IV parameter

Secret key and nonce is used to decrypt. Secret key is received from the recipient context

Code: POST OR FETCH (Just a dummy code) - 0.02

Token: Random hex - 0xFF

OSCORE: [Partial IV, kid context, kid parameter]
REMOVED
Uri-Host Outer: 1
...
Payload: **Plaintext**

6. Decrypt the COSE Object

7. Add Decrypted Code, and payload. OSCORE option is removed.

Code: **Decrypted - 2.04**

Token: Random hex - 0xFF
REMOVED
Uri-Host Outer: 1
...
Payload: Plaintext

8. Done!

Verifying OSCORE message → Generate CoAP Response → Protect CoAP Response

1. Retrieve Sender Context (Using sender ID) for target resource/recipient

2. Generate Additional Authenticated Data

3. Generate plaintext

- Request Code(type)
- Options
- Payload

(4. a) Encode the Partial IV, increase sender sequence number. Compute a new AEAD nonce

(4. b) Use the AEAD nonce from the request

5. Encrypt the COSE Object with sender key.

Compress the object afterwards.

Include the newly constructed Partial IV

5. Encrypt the COSE Object with sender key.

Compress the object afterwards.

6. Format the CoAP message to OSCORE by adding the OSCORE message option

7. Send the message!

1. Receive the message

2. Discard all class E options (If a option that should be encrypted is visible in the header, we instead discard it)

3.
- Use the token parameter to receive a recipient context.

4. Compose the AAD

5. Compute the AEAD Nonce
- If partial IV is not present, the nonce from the request is used

Else compute nonce from partial iv, recipient id, and common iv

6. Decrypt the COSE Object

7. Add Decrypted Code, and payload. OSCORE option is removed.

8. Done!

Verify CoAP Response

6.2 An example ProVerif file

The ProVerif modelling language is explained in Chapter 3. In order to showcase the functionality explained we illustrate ProVerif by example, which is the example model seen below.

It is a model of two hosts using their asymmetric keys to agree on a symmetric key. The protocol supposedly holds forward secrecy in the case that the private key of host B is disclosed to an attacker. The queries in line 40-42 verifies that this is the case, and that forward secrecy does not hold in the scenario where the private key of host A is disclosed to an attacker.

```
1  free c: channel.
2  type host.
3  type key.
4  type pkey.
5  type skey.
6  free A, B:host.
7  fun pk(host) : pkey.
8  fun sk(pkey) : skey [private].
9  fun sign(bitstring, skey): bitstring.
10 reduc forall m: bitstring, pubkey: pkey; getmess(sign(m, sk(pubkey))) = m.
11 reduc forall m: bitstring, pubkey: pkey; checksign(sign(m, sk(pubkey)), sk(pubkey))
       = m.
12
13 free s: bitstring[private].
14 fun aenc(bitstring, pkey) : bitstring.
15 reduc forall x:bitstring, pubkey:pkey; adec(aenc(x, pubkey), sk(pubkey)) = x.
16 fun enc(bitstring,  key) : bitstring.
17 reduc forall x: bitstring, k:key; dec(enc(x, k), k) = x.
18
19 let clientA(pkA: pkey) =
20   in(c, x:bitstring);
21   let y = adec(x, sk(pkA)) in
22   let (=pkA, =pk(b), k:key) = checksign(y, sk(pk(b))) in
23   out(c, enc(s, k)).
24
25 let clientB(pkB: pkey) =
26   in(c, pkX: pkey);
27   new k: key;
28   out(c, aenc(sign((pkX, pkB, k), sk(pkB)), pkX)).
29
30 query attacker(s) phase 0;
31       attacker(s) phase 1;
32       attacker(s) phase 2.
33
34 process
35   let pka = pk(a) in
36   let pkb = pk(b) in
37   clientA(pka) | clientB(pkb)   |
38   (phase 1; out(c, sk(pk(b)))) |
39   (phase 2; out(c, sk(pk(a))))
```

6.3 The horn clauses of ProVerif

The Horn clauses derived in ProVerif are explained in Chapter 3. The example ProVerif model in Appendix 6.2 was used to showcase some of the Horn clauses generated by ProVerif, for which a snippet can be seen below. The purpose of the Appendix is to showcase how the model has been translated from the Applied-Pi calculus model into Horn clauses and the capabilities of the attacker.

```
1 ...
2 attacker(v_44) -> attacker(pk(v_44))
3 attacker(true)
```

```
attacker_p1(v_145) -> attacker_p1(pk(v_145))
attacker_p1(sign(m_147,sk(pubkey_148))) -> attacker_p1(m_147)
attacker_p1(false)
attacker_p1(v_152) && attacker_p1(v_153) -> attacker_p1(enc(v_152,v_153))
attacker_p1(enc(x_154,k_155)) && attacker_p1(k_155) -> attacker_p1(x_154)
attacker_p1(v_168) && attacker_p1(v_169) -> attacker_p1(aenc(v_168,v_169))
attacker_p1(aenc(x_170,pubkey_171)) && attacker_p1(sk(pubkey_171)) -> attacker_p1(
    x_170)
attacker_p1(v_183) && attacker_p1(v_184) && attacker_p1(v_185) -> attacker_p1((
    v_183,v_184,v_185))
attacker_p1((v_186,v_187,v_188)) -> attacker_p1(v_186)
attacker_p1((v_190,v_191,v_192)) -> attacker_p1(v_191)
attacker_p1((v_194,v_195,v_196)) -> attacker_p1(v_196)
attacker_p1(v_200) && attacker_p1(v_201) -> attacker_p1((v_200,v_201))
attacker_p1((v_202,v_203)) -> attacker_p1(v_202)
attacker_p1((v_205,v_206)) -> attacker_p1(v_206)
attacker_p1(v_209) -> attacker_p1((v_209))
attacker_p1((v_210)) -> attacker_p1(v_210)
mess_p1(v_213,v_212) && attacker_p1(v_213) -> attacker_p1(v_212)
attacker_p1(v_215) && attacker_p1(v_214) -> mess_p1(v_215,v_214)
attacker_p1(fail-any)
attacker(v_217) -> attacker_p1(v_217)
table(v_218) -> table_p1(v_218)
attacker_p2(true)
attacker_p2(v_225) && attacker_p2(v_226) -> attacker_p2(sign(v_225,v_226))
attacker_p2(v_228) -> attacker_p2(pk(v_228))
...
```

6.4 The workaround made to the main process

As explained in Chapter 4.5.1, certain edits were made to the main process of the model in order to derive a result from the queries. This is the edited main process, and it allows us to derive a fake attack for both queries.

```
process
    in(c, initiator: host);
    in(c, responder: host);
    new IDir:id; new IDri:id;
    new msecret: bitstring; new msalt: bitstring; new idcontext: bitstring;
    out(c, (IDir, IDri, idcontext));
    insert security_context_lookup(initiator, responder, IDir, IDri, msecret, msalt
    , idcontext);
    insert security_context_lookup(responder, initiator, IDri, IDir, msecret, msalt
    , idcontext);
    (*  (!init_keys) | *)
    (!oscore_initiator) |
    (!oscore_responder) |
    (!oscore_initiator_response_receiver)
```