Formal verification of the PSP Security Protocol

Kwan Loong Chan

Master of Science
Cyber Security, Privacy and Trust
School of Informatics
University of Edinburgh
2024

Abstract

Formal verification offers a robust framework for proving the security of protocols or identifying potential attacks. These methods have been effectively used in the design and analysis of industrial protocols. In this project, we perform a formal verification of the PSP Security Protocol (PSP), developed by Google for encrypting data in transit between their data centres. This protocol was open sourced in 2022 for adoption, but as its intended security goals of confidentiality and authenticity have yet to be verified by the research community, this leaves open the possibility of implementations that contain hidden vulnerabilities.

We model and evaluate PSP in the symbolic model with the help of ProVerif, a tool that can automatically prove security properties when given as input a description of the protocol. Using ProVerif, we verify that the protocol is able to transmit a data packet from a transmitter to a receiver correctly, even when multiple connections are running concurrently. PSP makes use of the CMAC and GCM algorithms for which we have also developed models. These algorithms carry out critical cryptographic operations in PSP, such as encryption and authentication. Our models for these algorithms have been designed to accept input messages with arbitrary number of blocks so that data packets with arbitrary sizes can be supported in PSP. We formally verify these algorithms by treating them as separate models before integrating them into the main PSP model. We then show that all the security guarantees of PSP are satisfied, while noting that some of our proofs are only valid under certain conditions.

Besides proving the main security properties of PSP, we conduct a thorough study of known weaknesses in PSP and show how they can be handled by upper layer software. We also review alternative threat models such as compromised devices to see how they may impact PSP security.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Kwan Loong Chan*)

Acknowledgements

I would like to thank my supervisor, Dr Myrto Arapinis, for her time and guidance throughout the course of this project. Our weekly discussions were extremely helpful in broadening my understanding of the subject matter. Her insightful feedback on my findings also often provided me with fresh ideas to deepen my research into specific areas of work.

Table of Contents

1 Introduction **1**

1.1 Contributions 2

1.2 Dissertation structure 3

2 Background **4**

2.1 Symbolic and computational models 4

2.2 ProVerif 5

2.3 PSP security protocol 6

2.3.1 CMAC algorithm 7

2.3.2 GCM algorithm 8

2.4 Related work 9

3 Formal verification of CMAC **10**

3.1 Model design and implementation 10

3.1.1 Data authenticity 13

3.1.2 EUF-CMA 14

3.2 Results and analysis 14

4 Formal verification of GCM **16**

4.1 Model design and implementation 16

4.1.1 Data confidentiality and authenticity 18

4.1.2 IND-CCA2 18

4.2 Results and analysis 19

5 Modelling PSP **22**

5.1 Master key generation 22

5.2 SA generation and distribution 23

5.3 NIC Transmit 24

5.3.1 Egress classifier 24
5.3.2 Splitter 25
5.3.3 IV generator and AES GCM (encrypt) 25
5.3.4 Merge 26
5.4 NIC Receive 27
5.4.1 Ingress classifier 27
5.4.2 Splitter 27
5.4.3 Decryption key derivation, AES GCM (decrypt) and ICV check 28
5.4.4 Merge 29

6 Evaluation of PSP model **30**
6.1 Confidentiality of payload 30
6.2 Authenticity of payload 31
6.3 Uniqueness of connection parameters 32
6.4 Isolation between connections 33
6.5 Impact of compromised devices 34
6.6 Replay attack and defence 35
6.7 Bad-data-injection attack and defence 36
6.8 Concurrency 38

7 Conclusion **39**
7.1 Future work 40

Bibliography **41**

A ProVerif syntax **45**
A.1 Example program 45
A.2 List of syntax used in project 47

B Supplementary code in NIC Transmit and Receive processes **49**
B.1 NIC Transmit 49
B.1.1 Egress classifier 49
B.1.2 Splitter 50
B.1.3 Merge 51
B.2 NIC Receive 51
B.2.1 Ingress classifier 51
B.2.2 Splitter 52

B.2.3 Merge 53

C Supplementary queries in PSP model **54**

C.1 Secrecy queries and results 54

C.2 Queries to test isolation between connections 55

C.3 Queries and trace graphs for concurrency tests 57

C.3.1 Consecutive transmissions within same connection 57

C.3.2 Concurrent connections using same master keys but different SA keys 57

C.3.3 Concurrent connections using master keys 0 and 1 on same receiver 58

C.3.4 Concurrent connections using different master keys on different receivers 59

D Complete list of programs and results **61**

Chapter 1

Introduction

Security protocols are distributed programs which ensure the security of communications over a public network. Examples of security protocols that are widely used today include Transport Layer Security (TLS) for establishing a secure connection to a web site over the internet, and the Signal Protocol for enabling end-to-end encryption in instant messaging applications such as WhatsApp [1]. As these protocols are usually deployed over untrusted networks, it is important for their security properties to be guaranteed when data is being transmitted between the endpoints.

PSP Security Protocol (PSP) was developed by Google to protect data in transit between their data centres located throughout the world [2]. It is a custom-built protocol that is functionally similar to Internet Protocol Security Encapsulating Security Payload (IPsec ESP) but has been simplified to retain only the relevant features. This allows PSP to efficiently support per-connection encryption and authentication of large-scale traffic without incurring high latency [3]. The protocol also supports the offloading of these operations to the network interface card (NIC) so that they would not consume significant resources on the host servers.

After deploying and using PSP on their servers for around a decade, Google decided to release the architecture specification to the public in 2022 to encourage wider adoption [3, 4]. However, to the best of our knowledge, there has not been any published research to verify PSP's intended security properties of data confidentiality and authenticity. If the design of PSP contains any security flaw, then implementations of the protocol would also be vulnerable. This could then be exploited by a malicious adversary, resulting in users' data being compromised.

This project aims to fill in this gap by analysing the security of PSP using formal verification techniques. Formal methods provide a rigorous mathematical framework

that allows one to precisely describe the capability of the adversary and the security guarantees of the protocol [5]. We will make use of the symbolic model, also known as the Dolev-Yao model [6], where cryptographic primitives are represented by perfect black boxes with their operations described by functions and equations [7]. Under this model, the adversary is assumed to have control over the public network and can inspect or tamper with publicly transmitted messages, but he is only allowed to perform computations using those primitives.

The formulation of the symbolic model has led to the development of tools that can analyse security protocols with varying degree of automation [8]. For this project, the modelling of PSP and evaluation of its security will be done using ProVerif [9]. ProVerif can automatically prove security properties such as secrecy and authentication when given as input a description of the protocol in the applied pi calculus [7]. Researchers have successfully used ProVerif to analyse and discover attacks in real-world protocols such as TLS 1.3 [10, 11]. If any vulnerabilities in PSP are uncovered from our analysis, we will propose solutions to mitigate them and prove that the fixes are valid.

1.1 Contributions

Our main contributions in this work are as follows:

1. We develop a symbolic model of the specification of PSP written in ProVerif that consists of all its components and operations. This includes finer-grain operations that are not found within the PSP specification but are present in Google's reference implementation in C [12]. We also model the roles of the transmitter and receiver, showing how a data packet is encrypted and decrypted.

2. We develop and test symbolic models for two components that carry out the key cryptographic operations in PSP.

   (a) **Cipher-based message authentication code (CMAC)** algorithm [13] used during derivation of the encryption/decryption keys

   (b) **Galois/Counter Mode (GCM)** of operation [14] for the Advanced Encryption Standard (AES) algorithm used during packet encryption and authentication

   We have modelled the detailed operations within each algorithm instead of modelling them as black boxes. Our models are modular and easy to interface with.

We have packaged them as ProVerif library files and integrated them into the main PSP model, which greatly extends its coverage and precision.

3. We provide a novel way of modelling messages which consist of more than one block during CMAC and GCM operations, using iterative-like methods to process messages with unbounded lengths. This is a variation of the technique used by [15] and it enables us to create interesting security tests that involve messages with multiple block sizes.

4. We prove the specified security requirements of confidentiality and authenticity of the transmitted data in our PSP model in the presence of a Dolev-Yao adversary. We further show that the transmitter and receiver agree on a unique set of connection parameters and that the connections are isolated from one another.

5. We review known weaknesses that have been documented in PSP specification and show how they can lead to security problems. This includes replay attacks and bad data injection. We then model the recommended fixes at the upper layer of the network stack to mitigate these issues. Besides these known weaknesses, our analysis did not uncover other vulnerabilities in the symbolic model of PSP.

6. We analyse alternative threat models such as incorrect implementations or malicious principals, and also demonstrate how a more powerful adversary can undermine the secrecy and authenticity of the transmitted data.

We have submitted our ProVerif programs, their results, and a README file that explains each file and their functions. The results can be reproduced by running the programs on ProVerif 2.05, which is the latest version at the time of writing. We also provide the full listing and description of the files in Appendix D for reference.

1.2 Dissertation structure

The remainder of this dissertation is organised as follows. In Chapter 2, we review existing literature and provide justifications for our research methodology. We also describe PSP and its security goals. Chapters 3 and 4 respectively describe and analyse our CMAC and GCM models. Where relevant, we also point out the limitations of our design. In Chapter 5, we discuss our model for PSP in detail, followed by Chapter 6 where we evaluate PSP security and demonstrate some potential attacks. We conclude in Chapter 7 and suggest some work for further exploration.

Chapter 2

Background

2.1 Symbolic and computational models

In order to derive rigorous security proofs, protocols often need to be modelled mathematically using one of two approaches: the symbolic or computational model [16]. This section briefly explains each approach and our reasons for choosing the former.

The symbolic model is an abstract model that enables automated verification of protocol specifications using specially built tools, leading to increased accuracy and allowing for more attack vectors to be tested within the same time period. This model was pioneered by Needham and Schroeder in 1978 [17] and formalised by Dolev and Yao in 1983 [6]. Cryptographic primitives are modelled as perfect black boxes and their operations can be represented by functions and equations. For instance, symmetric key encryption can be modelled using function symbols **enc** and **dec**, where $\mathbf{enc}(k,m)$ represents the encryption of m with key k, and $\mathbf{dec}(k,c)$ represents the decryption of c with key k. The decryption operation can then be further described using the equation: $\mathbf{dec}(k,\mathbf{enc}(k,m)) = m$, which provides the capability for one to recover m from $\mathbf{enc}(k,m)$ if he possesses key k. As these operations are also available to an adversary who is assumed to have control over the public network, any inadvertent leakage of key k would compromise the secrecy of m. An advantage of the symbolic model is that any discovered attack also holds in the computational model and therefore translates to a practical attack. Under additional assumptions on computational soundness, proofs of security in the symbolic model also implies the same for the computational model [7].

On the other hand, the computational model reflects actual execution of protocols more closely and allows a larger class of attacks to be studied. This model is generally used by cryptographers to formulate proofs of security with respect to a computationally

bounded adversary. Under this model, cryptographic primitives are treated as probabilistic algorithms over bitstrings, and proofs of security often involve reasoning about the advantage of the adversary being negligible [16]. Due to the higher complexity of analysis, security proofs in the computational model are difficult to automate, require extensive manual effort, and may be susceptible to human error.

While both the symbolic and computational models have their strengths and limitations, the ability to automate the discovery of practical attacks within a tight dissertation schedule makes symbolic analysis the method of choice for this project.

2.2 ProVerif

Besides ProVerif, there exists several other tools that can be used to analyse security protocols in the symbolic model, each with different strengths and trade-offs. For instance, Maude-NPA [18, 19] and Tamarin [20] support equational theories such as associativity and commutativity but may require longer verification time or user intervention. Scyther [21] is a fully automated tool that can prove secrecy and authentication properties, but it only supports a limited set of cryptographic primitives [7]. In contrast, ProVerif is fully automatic, supports an unbounded number of sessions of the protocol, and handles a wide variety of cryptographic primitives. It can be used to prove reachability, correspondences, and equivalence [16], which allow us to express security properties that are relevant to PSP. Although ProVerif may not always terminate and it makes abstractions that may result in false attacks, it has a proven track record of successfully being used to analyse many real-world protocols, such as TLS [10, 22]. These benefits outweigh the limitations and justify ProVerif as a suitable tool for this project.

The ProVerif user manual [23] contains an extensive suite of syntax and semantics, but we will only be using a subset of them. We have written an example program and provided a line-by-line explanation in Section A.1 of Appendix A. We also list the syntax used in our project in Section A.2 of Appendix A for reference.

In the background, ProVerif actually abstracts the program and the security properties into Horn clauses and derivability queries on those clauses [7]. A security property is true if it is not derivable by the adversary. If a security property is false, ProVerif will try to reconstruct an execution trace that shows how an attack on the property is achieved. There is also a chance of obtaining false attacks or non-termination due to the abstraction into Horn clauses. When this issue is encountered, we would manually inspect the output to diagnose the problem and refine our model to resolve it.

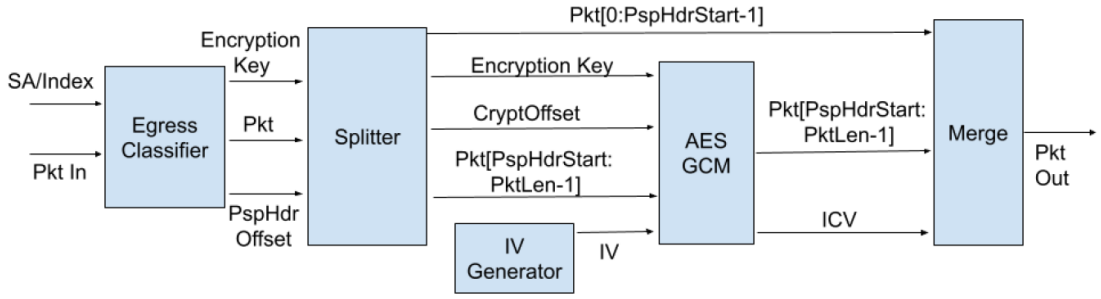

Figure 2.1: Transmitter block diagram [4].

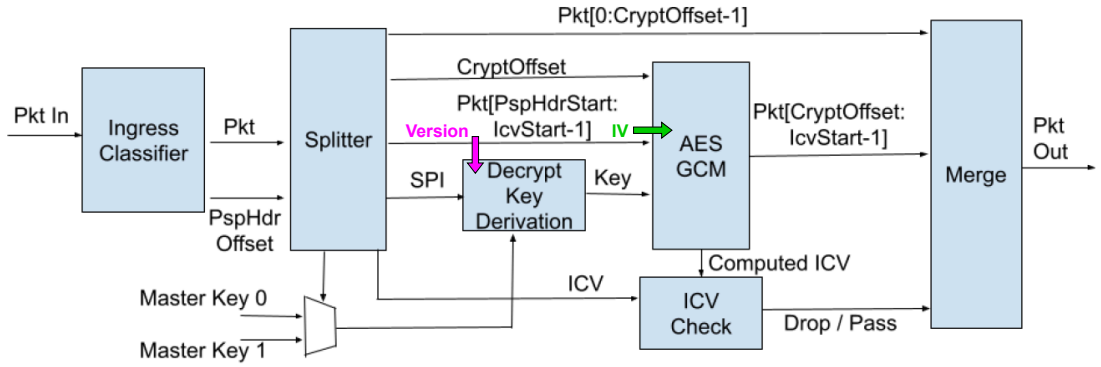

Figure 2.2: Receiver block diagram [4] with some details added.

2.3 PSP security protocol

PSP was developed by Google to provide confidentiality and authenticity of Internet Protocol (IP) packets at the network layer. Similar to IPsec ESP, PSP supports transport and tunnel modes of operation. Our research focuses on transport mode, but the models can be easily modified to work for tunnel mode, which only requires an additional header. Figures 2.1 and 2.2 are taken from the PSP specification and they provide an overview of the operations that occur at the transmitter and receiver side to secure the data. To improve the accuracy of our model, we studied the official C reference implementation of PSP and added any missing but important details. For instance, the Version field should be extracted from the PSP header and used for derivation of the decryption key (added in magenta). The C implementation also extracts the initialisation vector (IV) field from the PSP header and uses it to construct the IV for AES GCM decryption (added in green). Both of these fields had been abstracted into the Pkt[PspHdrStart:IcvStart-1] term in the original diagram.

As the key exchange protocol is outside the scope of PSP, we assume the transmitting party can request and obtain an encryption key from the receiving party prior to the start of PSP. Each receiver NIC stores a pair of 256-bit master keys, of which only one

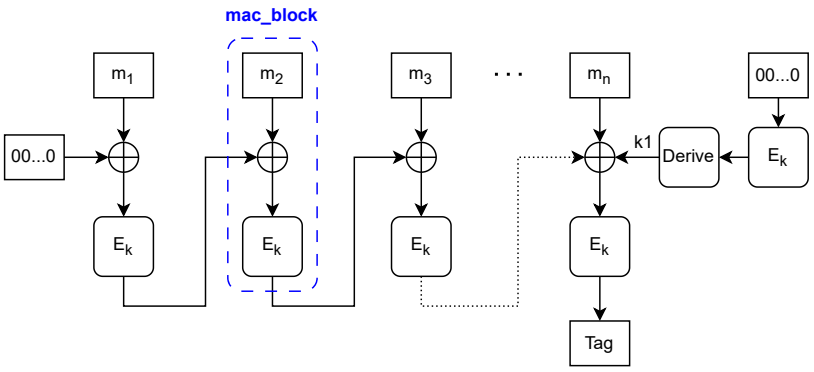

Figure 2.3: CMAC operation.

is active at any time. Upon receiving a request for a new encryption key, the receiver would generate one by running CMAC algorithm on some input data and its own active master key. The encryption key is then used to encrypt and authenticate data packets with AES GCM algorithm. Being a symmetric key algorithm, the decryption key in AES GCM is the same as the encryption key. The receiver does not need to store a copy of the decryption key as it can be derived from the PSP header sent by the transmitter, together with the receiver's own active master key. When all possible encryption keys for the active master key have been used up or after an expiry time, the other master key will become active while the old one will be replaced with a fresh random value.

The CMAC and GCM operations will be modelled and verified in detail in Chapters 3 and 4 as these are critical cryptographic operations in PSP. Here, we provide a brief overview of these algorithms in the following sub-sections.

2.3.1 CMAC algorithm

In general, CMAC is used to provide assurance of data authenticity. An overview of its operation is shown in Figure 2.3. The input message is first broken up into n blocks of 128 bits each. Each message block is an input into a routine which we have labelled as "mac_block" and will be referencing it in Chapter 3. The output from the previous block cipher with key k, denoted by E_k, will undergo exclusive-or (XOR) with the next input message block before the result is encrypted. The final message block is also XOR'ed with a subkey k1 that is derived from the encryption of the all-zeros block. The original CMAC specification contains a few subkeys, but only k1 is applicable to PSP. The output from the final block cipher is the authentication tag and any slight modification of the original message is expected to result in a drastically different tag.

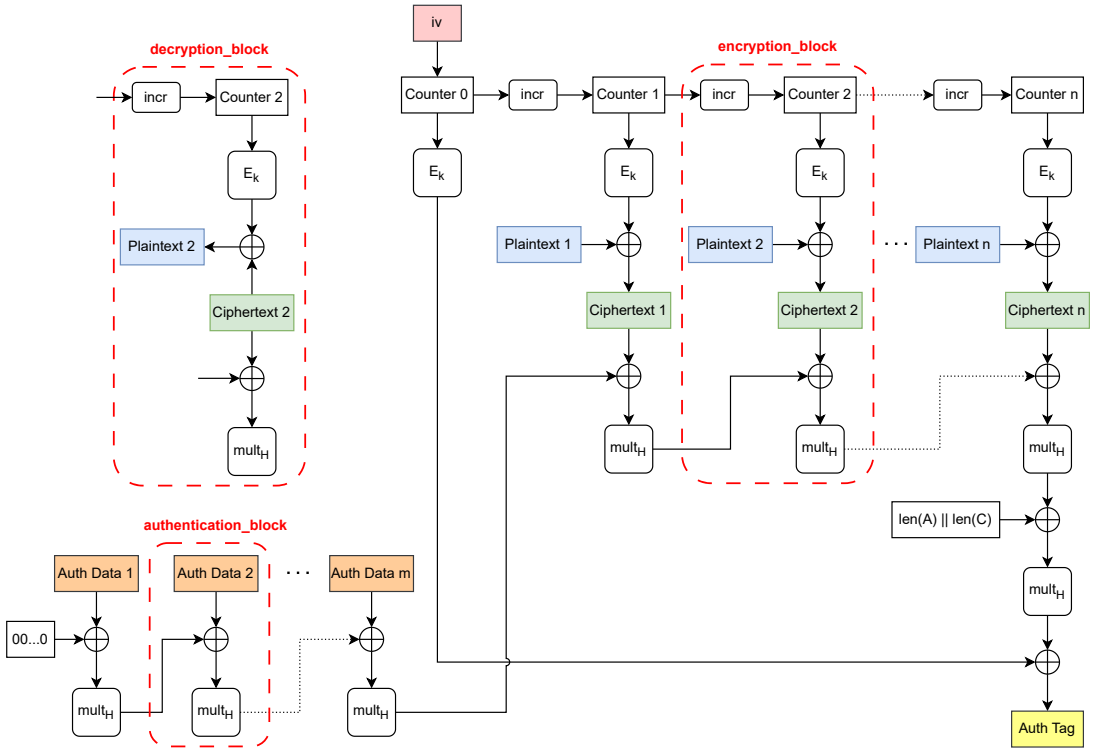

Figure 2.4: GCM operation.

2.3.2 GCM algorithm

GCM provides authenticated encryption of input data, which consists of the plaintext that needs to be encrypted and authenticated, and the additional authenticated data that only needs to be authenticated. Figure 2.4 provides an overview of its operation. We shall assume for simplicity that the input message has already been padded to contain exactly m Auth Data blocks and n Plaintext blocks of 128 bits each.

Encryption flow: A randomly generated IV is used to derive a counter, which is then incremented by one for each Plaintext block. This counter is encrypted using a block cipher with key k, denoted by E_k, and the result undergoes XOR with each Plaintext block to obtain the corresponding Ciphertext block.

Authentication flow: Each Auth Data block undergoes XOR with the result from the previous multiplication operation for the binary Galois field of 2^{128} elements [14], indicated by the $mult_H$ component. This operation uses a hash key H that is derived from key k. After all Auth Data blocks have been processed, a similar operation occurs for all the Ciphertext blocks that were generated from the encryption flow. At the end of the algorithm, the lengths of the additional authenticated data, ciphertext, and the output of the first block cipher are involved in deriving the authentication tag (Auth Tag).

We have grouped together some components into “encryption_block”, “decryption_block” and “authentication_block” in Figure 2.4, which will be referenced in Chapter 4. Decryption is similar to encryption, with all the encryption_blocks replaced by the decryption_blocks (note the reverse direction of arrows to recover Plaintext from Ciphertext). The sender needs to send the IV, additional authenticated data, ciphertext and Auth Tag to the receiver so that he is able to authenticate and decrypt the data.

2.4 Related work

Although formal verification of PSP has not been done before, there exists similar studies on security protocols that are widely used today, such as TLS 1.3 and the Signal Protocol. TLS 1.3 is used to secure communications to a web sites over the internet. Arai et al. used ProVerif to model the full handshake protocol and showed that it fulfilled its secrecy and authentication goals [24, 25]. Bhargavan et al. also used ProVerif to study different modes of operation in TLS 1.3 and covered various threat models [10]. The Signal Protocol has been deployed in many popular instant messaging applications, such as WhatsApp and Facebook Messenger. Kobeissi et al. used ProVerif to find new and known weaknesses in the protocol and suggested practical countermeasures [26].

The CMAC and GCM algorithms are recommended by NIST for data authentication and encryption [13, 14]. They have been extensively analysed for their performance and security properties. For instance, Black and Rogaway proved that an adversary is unable to forge a new tag using the CMAC algorithm [27], while Iwata and Kurosawa proved its security for arbitrary length messages [28]. The performance and security properties of GCM have also been analysed in [29] and [30]. We highlight that these studies had been carried out in the computational setting, whereas our work focuses on the formal verification of these algorithms using the symbolic model.

Mieno et al. introduced a technique to formalise loop iterations in protocols using ProVerif, treating function calls as communications between internal processes [15]. We adapted their technique and made some modifications to model messages containing multiple blocks. Instead of comparing block numbers to match the parent and child processes, we utilised fresh private reply channels to ensure that message replies from the child process would only be accepted by its parent without needing comparison operations. We also combine the Ciphertext blocks into a single term before returning it to the requestor. This way, the requestor would only need to transmit a single ciphertext to the receiving party instead of having to send multiple Ciphertext blocks.

Chapter 3

Formal verification of CMAC

In this chapter, we describe our model for the CMAC algorithm and analyse its security properties using ProVerif. Recall that CMAC is a critical component used in PSP to derive the encryption/decryption key. Therefore, verification of the CMAC model would directly strengthen our proofs of security for the PSP model.

3.1 Model design and implementation

We observe from Figure 2.3 of Section 2.3.1 that the CMAC operation is composed of iterations of the mac_block component. Each mac_block takes in an input message block, the previous encryption result, key k, and outputs the current encryption result to the next mac_block. Figure 3.1 and the accompanying code block illustrate our top-level model design, which consists of an unbounded number of CMAC and mac_block processes that are automatically generated based on the number of requests and input message blocks, respectively. All communications are carried out using private channels, so any transmitted message between these processes are not available to the adversary.

```
let CMAC_processes = (!CMAC) | (!mac_block).
```

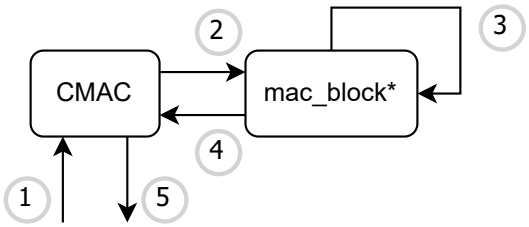

Figure 3.1: CMAC model with unbounded number of mac_blocks.

The steps in our implementation of the CMAC model are described below, and they correspond to the labels in Figure 3.1. The full program has been packaged into a library file named cmac.pvl so that it can be easily integrated into other larger models.

1. The CMAC process accepts the inputs from the requestor via a private channel `c_cmac`. The input comprises a fresh private reply channel `c_return`, key `K`, message `M`, and the number of message blocks `M_blocks`.

```
in(c_cmac, (c_return: channel, K: mkey_t, M: bitstring,
   M_blocks: nat));
```

2. The CMAC process creates a fresh private reply channel `c_return_mac`, before sending the received inputs, `c_return_mac` and the all-zeros block to the mac_block process via another private channel `c_mac_block`.

```
new c_return_mac: channel;
out(c_mac_block, (c_return_mac, K, M, M_blocks, ZEROS));
```

3. The mac_block process is divided into two parts separated by an if-then-else statement. If the current number of message blocks is 1, the computation is performed for the final message block as shown in Figure 2.3. The `menc` function first encrypts the all-zeros block and passes the result to the `derive_subkey` function. The `derive_subkey` function then outputs subkey `K1`. This operation is abstracted as it involves bit manipulation which cannot be modelled by ProVerif. This is followed by a series of `xor` and `menc` operations to return the final `mac`.

```
fun menc(mkey_t, bitstring): bitstring.
reduc forall k: mkey_t, m: bitstring; mdec(k, menc(k, m)) = m.
fun derive_subkey(bitstring): bitstring.
fun xor(bitstring, bitstring): bitstring
equation forall x: bitstring, y: bitstring; xor(xor(x,y),y) = x
```

```
let L = menc(K, ZEROS) in
let K1 = derive_subkey(L) in
let message' = xor(message, K1) in
let mac = menc(K, xor(message', prev_mac)) in
out(prev_c_return_mac, mac)
```

XOR properties are defined using a function and equation, which together provides the ability to recover operand `x` from `x`$\oplus$`y`. We are unable to model commutativity and associativity properties of `xor` as they are not supported by ProVerif.

ProVerif is also unable to recover both operands at the same time. We acknowledge that the absence of these properties may weaken the adversary's capability, and therefore any proofs of security would only apply under these conditions.

When the current number of message blocks is more than 1, we extract the first message block `curr_message` and compute the output of the current mac_block `curr_mac`. We then create another fresh private reply channel `c_return_mac` and send these inputs together with the remaining message and remaining number of message blocks into the `c_mac_block` channel. This triggers the generation of a child mac_block process to accept these inputs and repeat the computation under a new context. The current mac_block process then waits for the return value `mac` from the child process, before sending this value back to its parent.

```
let curr_message = head_block(message) in
let remaining_message = tail_blocks(message) in
let curr_mac = menc(K, xor(curr_message, prev_mac)) in
new c_return_mac: channel;
out(c_mac_block, (c_return_mac, K, remaining_message,
   message_blocks - 1, curr_mac));
in(c_return_mac, mac: bitstring);
out(prev_c_return_mac, mac)
```

The following set of equations define the behaviour for block splitting and combining, which are important operations for organising the message blocks.

```
equation forall m: bitstring; combine_blocks(head_block(m),
   tail_blocks(m)) = m.
equation forall head: bitstring, tail: bitstring; head_block(
   combine_blocks(head, tail)) = head.
equation forall head: bitstring, tail: bitstring; tail_blocks(
   combine_blocks(head, tail)) = tail.
```

4. The parent CMAC process receives the final `mac` from the mac_block process via the private reply channel `c_return_mac` that was created in step 2. This ensures that it receives the `mac` from its immediate child mac_block.

```
in(c_return_mac, T: bitstring);
```

5. Finally, the CMAC process returns `mac` to the requestor via the private reply channel `c_return` that was created by the requestor in step 1.

```
out(c_return, T).
```

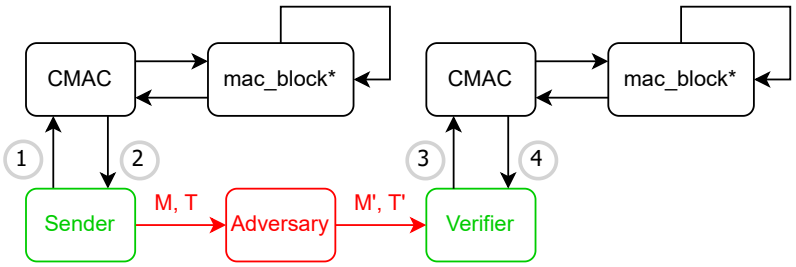

Figure 3.2: Sender-Verifier model.

3.1.1 Data authenticity

Data authenticity is the main security property that must be achieved by CMAC. This means that any unauthorised modification of the data must be detected. Figure 3.2 shows a typical Sender-Verifier model, where honest parties are highlighted in green and the adversary controls the transmission medium highlighted in red. The numbers denote private channels. In this model, only the sender and verifier possess knowledge of the CMAC key. Referring to Listing 3.1, the sender computes the authentication tag `T` for an arbitrary message `M` using the CMAC implementation from Section 3.1 and sends both `M` and `T` to the verifier. The verifier then recomputes the tag from the received message using the same CMAC implementation and checks whether the computed tag matches the received tag. If they do not match, then either `M` or `T` must have been modified. The full implementation for this model can be found in cmac_sender_verifier.pv.

```
1  let sender(K: mkey_t, M_blocks: nat) =
2      new M: bitstring;
3      new c_return: channel;
4      out(c_cmac, (c_return, K, M, M_blocks)); (* channel 1 *)
5      in(c_return, T: bitstring);              (* channel 2 *)
6      event message_sender(M, T);
7      out(c_public, (M, T)).
8
9  let verifier(K: mkey_t, M_blocks': nat) =
10     in(c_public, (M: bitstring, T': bitstring));
11     new c_return: channel;
12     out(c_cmac, (c_return, K, M, M_blocks')); (* channel 3 *)
13     in(c_return, T: bitstring);               (* channel 4 *)
14     if T = T' then
15     event message_verifier(M, T).
```

Listing 3.1: Sender-Verifier processes.

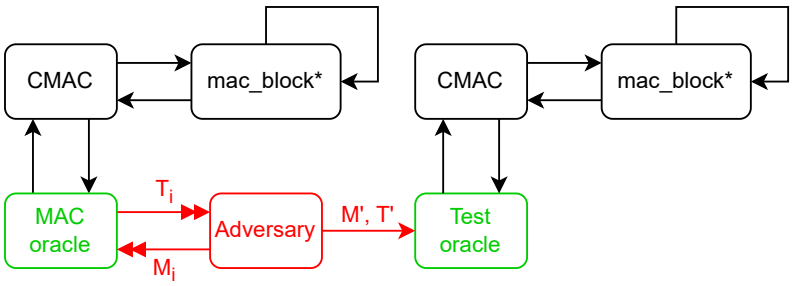

Figure 3.3: EUF-CMA model.

3.1.2 EUF-CMA

Existential unforgeability under adaptive chosen message attack (EUF-CMA) is another property that should be satisfied by a MAC system. This means that an adversary cannot construct a valid tag on any new message, even if the adversary can obtain tags on other arbitrary messages [31]. Our model to test this property is shown in Figure 3.3. The MAC oracle acts as a proxy for the adversary to access the CMAC implementation without knowing the key. He can submit requests for tags T_i of arbitrary messages M_i. The adversary then submits a message-tag pair (M', T') to a test oracle, where M' has not been queried to the MAC oracle before. The adversary is successful if T' is a valid tag for M', and $M' \notin \{M_i\}$. The full implementation can be found in cmac_euf-cma.pv.

3.2 Results and analysis

We placed event checkpoints at the end of the honest processes in the Sender-Verifier and EUF-CMA models and ran the following queries, with both producing true results.

```
query m: bitstring, t: bitstring; event(message_verifier(m, t)) ==>
    event(message_sender(m, t)). (* Sender-Verifier model *)
query m: bitstring, t: bitstring; event(message_test_oracle(m, t))
    ==> event(message_mac_oracle(m, t)). (* EUF-CMA model *)
```

A true result in the Sender-Verifier model means that whenever the verifier receives a valid message-tag pair (m,t), the same message and tag must have been generated by the sender, which implies that the message is authentic and has not been modified by the adversary. Similarly, a true result for the EUF-CMA model means that whenever the test oracle receives a valid message-tag pair (m,t), the same message must have been previously queried to the MAC oracle, which implies that a tag cannot be forged for any new message. Therefore, these results validate the accuracy and intended security

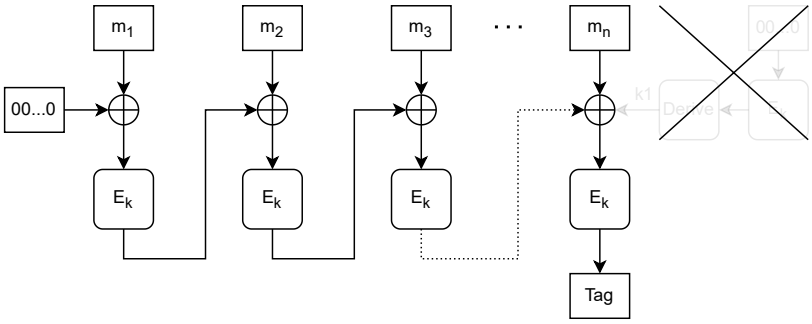

Figure 3.4: CMAC operation without subkey derivation.

properties of our CMAC model. The complete set of results in HTML form are archived in cmac_sender_verifier_results and cmac_euf-cma_results folders.

One challenge encountered during the analysis is that the resolution algorithm in ProVerif would not terminate if we allow an unbounded number of message blocks in the CMAC model. This issue also applies to the GCM model covered in Chapter 4. Our solution is to limit the maximum number of message blocks such that the analysis can complete within a reasonable time, and then show that the security properties are true under these constraints. We have proven data authenticity in the Sender-Verifier model for up to 6 input message blocks, which required 11 minutes to complete the analysis on a DICE compute server (selby.inf.ed.ac.uk). Proving this for higher number of blocks may require reduction or manual analysis due to limited computational power. Nevertheless, the recurring structure of the CMAC model provides good confidence that our current proof for 6 blocks would continue to hold for higher number of blocks.

To further validate the accuracy of our CMAC model, we compared it to one that has been incorrectly implemented. Figure 3.4 shows a flawed implementation of CMAC algorithm that has omitted the subkey derivation component. Using this implementation of CMAC in the Sender-Verifier model, ProVerif showed that the query for data authenticity was false and demonstrated a length-extension attack. In this attack, the adversary would intercept two single-block message-tag pairs $(M1, T1)$ and $(M2, T2)$ from the sender, and then forward a 2-block message-tag pair $((M1, M2 \oplus T1), T2)$ to the verifier. Since $T2$ is a valid tag for $(M1, M2 \oplus T1)$, the adversary has successfully forged a tag for a new message and the verifier unknowingly accepted the new message. The flawed CMAC implementation can be found in cmac_no_k1.pvl, while the results and trace graph are archived at cmac_no_k1_length_extension_attack folder. This demonstration clearly emphasises the importance of following the specifications properly when implementing algorithms.

Chapter 4

Formal verification of GCM

The GCM operation is used with the AES algorithm in PSP to carry out its main function: packet encryption and authentication. In this chapter, we review our implementation of the GCM model and show how its intended security properties are satisfied. The GCM architecture is more complex than CMAC, but there are some similarities. We will focus more on the new components which have not been covered in Chapter 3.

4.1 Model design and implementation

The iterative structure of the GCM operation allows us to break it up into recurring instances of encryption_block, decryption_block, and authentication_block, indicated by the red dashed boxes in Figure 2.4 of Section 2.3.2. Our top-level model designs are shown in Figure 4.1, along with the implementation code shown below. As before, the number of processes generated depends on the number of requests and input blocks specified. We can apply the same iterative technique used in Chapter 3 to model the GCM operation with arbitrary number of Auth Data, Plaintext, and Ciphertext blocks.

```
let GCM_processes = (!GCM_AE) | (!GCM_AD) | (!authentication_block)
   | (!encryption_block) | (!decryption_block).
```

We now describe the steps for encryption in Figure 4.1. The complete program has been packaged into a library file named gcm.pvl so that other models can use it.

1. The GCM Authenticated Encryption (GCM_AE) process waits for the requestor to send the required inputs. After completion of the entire process, the GCM_AE process returns ciphertext `C` and authentication tag `T` back to the requestor. The implementation is similar to that of the CMAC model.

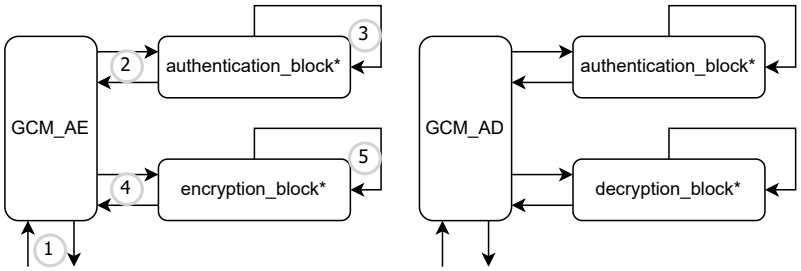

Figure 4.1: GCM encryption (left) and decryption (right) models with unbounded number of blocks.

2. The GCM_AE process uses the input `IV` to initialise `counter_0`. It also prepares the hash key `H` which is defined as the encryption of the all-zeros block using block cipher E_k. It then sends the required information to the authentication_block process using a similar implementation as the CMAC model, and waits for the result from processing all the Auth Data blocks.

```
let counter_0 = counter(IV, 0) in
let H = senc(K, ZEROS) in ...
```

3. The authentication_block process performs the computation for each Auth Data block, which is the term `curr_auth_data` in the code below. The code follows our description of the **authentication flow** in Section 2.3.2. This process also contains code (not shown) that would trigger the same process for the next Auth Data block, allowing for an unbounded number of such blocks to be processed.

```
let curr_mult_out = mult(H, xor(prev_mult_out, curr_auth_data))
    in ...
```

4. The GCM_AE process proceeds with the **encryption flow** in Section 2.3.2 by triggering the encryption_block process using the necessary inputs. At the end of encryption, it will receive the combined ciphertext `C` and intermediate authentication tag `final_mult_out` back from the encryption_block process. It then computes the final authentication tag using the lengths of the additional authenticated data and ciphertext, as well as the output from the first block cipher.

```
in(c_combine, (C: bitstring, final_mult_out: bitstring));
let T = xor(mult(H, xor(final_mult_out, (len(A), len(C)))),
   senc(K, counter_0)) in ...
```

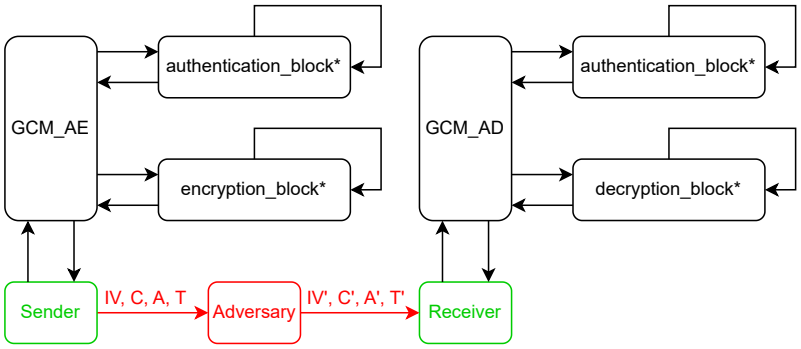

Figure 4.2: Sender-Receiver model.

5. The encryption_block process performs the computation for each Plaintext block to obtain the Ciphertext block, and authenticates the Ciphertext block using similar code as step 3. It also contains code (not shown) that would trigger the same process for the next Plaintext block, allowing for an unbounded number of Plaintext blocks to be processed. We combine all Ciphertext blocks back into a single ciphertext term before returning it to the requestor.

```
let combined_ciphertext = combine_blocks(curr_ciphertext,
    remaining_ciphertext) in ...
```

We now model the security properties of the GCM model in the presence of a Dolev-Yao adversary who is able to interact with the processes via the public channel.

4.1.1 Data confidentiality and authenticity

Figure 4.2 shows our model to test for data confidentiality and authenticity in GCM, where only the sender and receiver possess knowledge of the encryption/decryption key. The sender has some plaintext P and additional authenticated data A that he wishes to encrypt using the GCM algorithm. He generates a random IV, sends the inputs into the GCM encryption model, and receives back ciphertext C and authentication tag T. The sender transmits (IV, C, A, T) to the receiver, who then uses the GCM decryption model to successfully recover the plaintext P only if the authentication tag T is valid. The implementation for this model can be found in gcm_sender_receiver.pv.

4.1.2 IND-CCA2

Indistinguishability under adaptive chosen ciphertext attack (IND-CCA2) is a strong property of an encryption scheme and can be defined in the form of a randomised

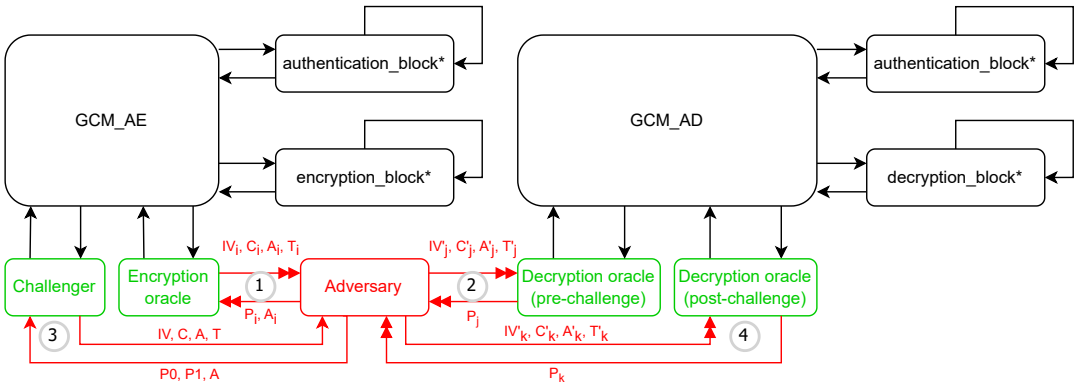

Figure 4.3: IND-CCA2 model.

experiment with a powerful adversary [31]. Figure 4.3 shows our model to test GCM for IND-CCA2, with numbered labels to identify the transactions. The adversary has oracle access to GCM encryption and decryption and is allowed to make any queries (labels 1 and 2). He then creates and sends two distinct but equal-length plaintexts to a challenger, who randomly chooses one of them to encrypt and returns the challenge ciphertext back to the adversary (label 3). The adversary continues to have oracle access to GCM encryption and decryption, but is not allowed to query the latter on the challenge ciphertext itself (labels 1 and 4). Eventually, the adversary needs to determine which of the two plaintexts has been encrypted by the challenger. We say that the encryption scheme is IND-CCA2 secure if the adversary has negligible advantage in distinguishing between the two plaintexts. The implementation for this model can be found in gcm_ind-cca2.pv and uses the "phase" instruction to arrange the order of steps.

4.2 Results and analysis

The properties of data confidentiality and authenticity were tested using queries on the secrecy of the payload and the reachability of events placed within the Sender-Receiver model. The result shows that these properties are satisfied by our GCM models. The HTML output for single Plaintext and Auth Data block has been archived at gcm_sender_receiver_results folder. We were able to test up to 2 Plaintext and 2 Auth Data blocks, which took 1h 40mins to complete on the DICE compute server (selby).

For IND-CCA2, we made use of a new syntax "choice[P0, P1]", which checks for observational equivalence between a process that uses plaintext P0 and a process that uses another plaintext P1. Below shows the code snippet in the challenger process, who

is sending either P0 or P1 to the GCM_AE process and receiving back ciphertext C and authentication tag T. If there is observational equivalence, then the adversary is not going to be able to tell whether C came from P0 or P1. Our test shows that there is indeed observational equivalence, which proves that our GCM model is IND-CCA2 secure. The HTML output is archived at gcm_ind-cca2_results folder. Unfortunately, due to the higher complexity of adversary interactions, we were only able to obtain the proof for a single Plaintext and Auth Data block. When setting a larger number of blocks, the analysis would not complete even after running for 24 hours.

```
out(c_gcm_enc, (c_return, K, IV, choice[P0, P1], P_blocks, A,
    A_blocks));
in(c_return, (C: bitstring, T: bitstring));
```

To further validate the accuracy of our GCM model, we compared it to two other models that were incorrectly implemented. Figure 4.4 shows a flawed implementation of GCM algorithm that has omitted the final XOR operation with encrypted Counter_0, which can be found in gcm_no_final_xor.pvl. Using this implementation of GCM in the Sender-Receiver model, ProVerif discovered a message forgery attack. The adversary intercepts (IV, C, A, T) from the sender and sends (IV', C, A, T) to the receiver. Tag T is still valid because it now only depends on the additional authenticated data A and ciphertext C, which are both unchanged. Receiver proceeds to decrypt the ciphertext using IV', resulting in a different plaintext. The trace graph can be found in the gcm_no_final_xor_message_forgery folder.

Figure 4.5 shows another flawed implementation which has omitted the length computation near the end of the algorithm. This is implemented in gcm_no_length.pvl. Using this implementation of GCM in the Sender-Receiver model, ProVerif discovered a length extension attack. The adversary intercepts a single block ciphertext and additional authenticated data (IV, C, A, T) from sender, and sends a 2-block ciphertext and a single block all-zeros authenticated data (IV, (A, C), zeros, T) to the receiver. Tag T is still valid because the first ciphertext block A behaves exactly like the authenticated data, and the lengths of the ciphertext and additional authenticated data are not being checked even though they have been modified. Therefore, the receiver proceeds to decrypt the ciphertext and obtains a different plaintext. The trace graph can be found in the gcm_no_length_length_extension_attack folder.

These two flawed models provide a clear picture of why certain features are required in the GCM algorithm. They also highlight the ability of ProVerif to automatically discover attacks that may not be obvious when designing new cryptographic algorithms.

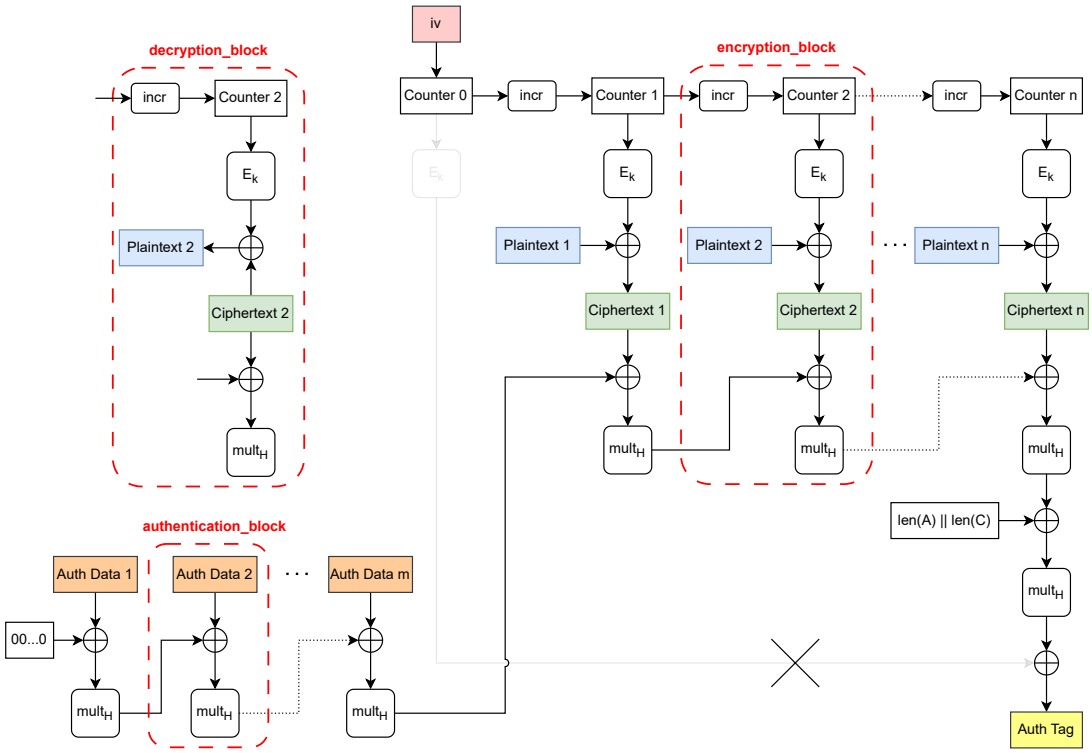

Figure 4.4: GCM model without final XOR operation.

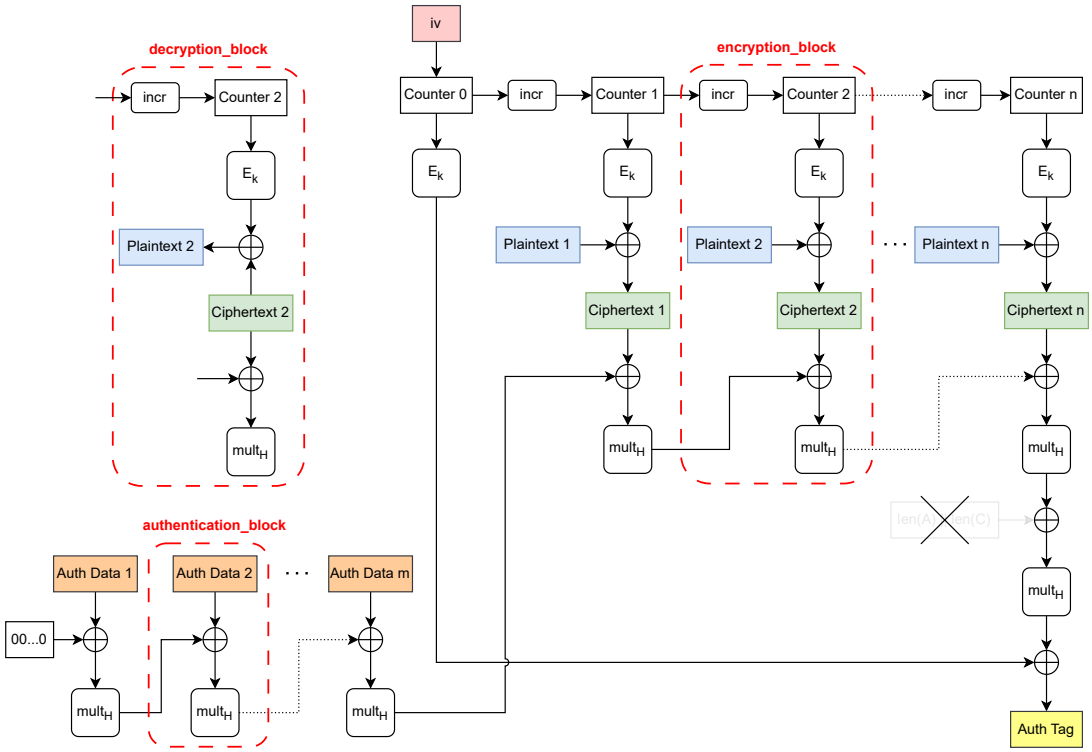

Figure 4.5: GCM model without length computation.

Chapter 5

Modelling PSP

We have so far developed detailed models for the CMAC and GCM algorithms and verified that their security properties are being achieved by our models. In this chapter, we describe our implementation for the PSP protocol and show how the CMAC and GCM models are integrated into this model. Our symbolic model for PSP is shown in Figure 5.1. It shows the key processes and how information propagates between them. The full program can be found in the file psp.pv. The code below shows the main process in the program launching an unbounded number of sessions of all processes and running them in parallel. `P_blocks`, `C_blocks` and `A_blocks` define the number of blocks in the input data, which are subsequently used by the CMAC and GCM models. These numbers can be modified, and we have tested the PSP model for up to 2 `P_blocks`, 2 `C_blocks` and 2 `A_blocks`. `CMAC_processes` and `GCM_processes` refer to the processes that have been presented in Chapters 3 and 4.

```
process
    let P_blocks = 1 in
    let C_blocks = 1 in
    let A_blocks = 1 in
    ( (!master_key_generation) | (!sa_generation_and_distribution) |
     (!NIC_Transmit(P_blocks, A_blocks)) | (!NIC_Receive(C_blocks,
   A_blocks)) | CMAC_processes | GCM_processes )
```

5.1 Master key generation

Each NIC receiver holds a pair of 256-bit master keys, of which only one of them is active at any one time and used to derive the secret key for encryption/decryption.

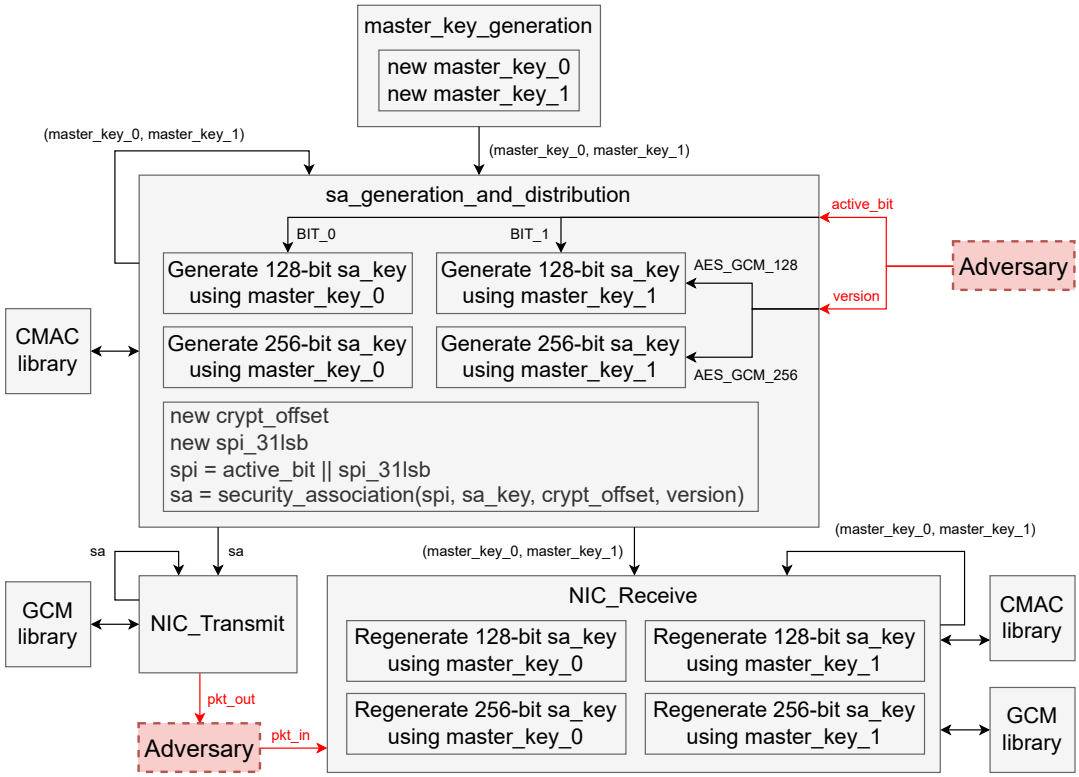

Figure 5.1: PSP model.

Each master key has an expiry period, after which the other master key becomes the active key while the current one will be regenerated using some strong and approved cryptographic technique. Our ProVerif code for the `master_key_generation` process simply generates a pair of master keys and sends them to the next process.

```
let master_key_generation =
    new master_key_0: mkey_t;
    new master_key_1: mkey_t;
    event end_master_key(master_key_0, master_key_1);
    out(c_sa, (master_key_0, master_key_1)).
```

5.2 SA generation and distribution

Although the initial handshake for key distribution is outside the scope of PSP, we still model the handshake in the form of a security association (SA) generation and distribution process, so that the secret parameters are guaranteed to be securely generated and distributed to the transmitter and receiver. This process resides on the receiver and uses the master keys from the previous section. The secret key for encryption/decryption

is generated using the CMAC library, with AES as the block cipher and the active master key as the CMAC key. The CMAC library takes in a single message block that is the concatenation of a random 32-bit security parameters index `spi` and three other 32-bit fields that depend on whether a 128-bit or 256-bit version of AES is required. The output of CMAC is the encryption key `sa_key`. Referring to Figure 5.1, we allow the adversary to determine which master key is active (`active_bit`) and what version of AES is selected (`version`). Giving the adversary more power not only strengthens our security proofs, but also allows us to test for all four scenarios indicated in the figure. The `sa_generation_and_distribution` block contains code snippets describing how the `spi` is constructed, where the `active_bit` becomes its most significant bit while the remaining 31 bits are random. The `spi`, `sa_key`, `crypt_offset` and `version` are packaged together as the `sa` before sending it to the transmitter via a private channel. There is also a feedback loop that sends the current master keys back to this process so that it can reuse the master keys to create SAs for different transmitters.

5.3 NIC Transmit

The transmitter receives the SA from the previous process and is ready to initiate the PSP protocol with the receiver. This process uses the GCM library for authenticated encryption. Note in Figure 5.1 that this process also contains a feedback loop that allows it to reuse the same SA for messages within the same connection. We will refer to the components shown in the block diagram of Figure 2.1 in Chapter 2 and describe our implementation of each component in this section. Some of the implementation code has been moved to Section B.1 of Appendix B, where they will be further explained.

5.3.1 Egress classifier

The egress classifier is responsible for building the PSP and UDP headers into the original IP packet. The PSP header contains information that is needed by the receiver to decrypt the packet, while the UDP header contains routing information that depends on inner header fields. The egress classifier also communicates the PSP header offset (from the start of the packet) to the splitter block. We have implemented the egress classifier as a destructor, and our code is further explained in Appendix B. Figure 5.2 illustrates how our implementation transforms the IP packet.

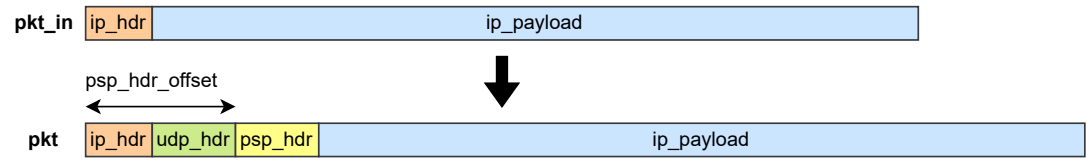

Figure 5.2: Transformation of IP packet by egress classifier.

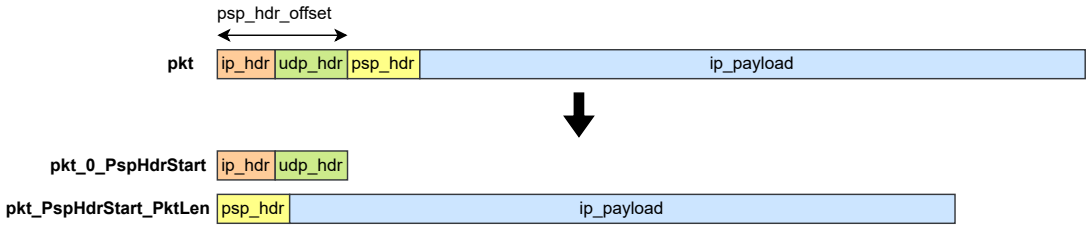

Figure 5.3: Division of data packet by splitter.

5.3.2 Splitter

The main function of the splitter is to divide the IP packet into two segments based on the PSP header offset received from the egress classifier. The segment from the start of the packet to just before the PSP header will be sent in clear, while the segment from the PSP header to the end of the packet will be sent to the AES GCM block. The splitter also determines the crypt offset from the PSP header and sends it to the AES GCM block. This crypt offset starts from the end of the PSP header and is used to determine the portion of IP payload that does not need to be encrypted but remains authenticated. We have implemented the splitter as a destructor and our code is further explained in Appendix B. Figure 5.3 illustrates how our implementation splits up the data packet.

5.3.3 IV generator and AES GCM (encrypt)

Figure 5.4 shows how the data packet will be transformed by the AES GCM block. The terms in this figure are taken from our implementation in Listing 5.1. A fresh random

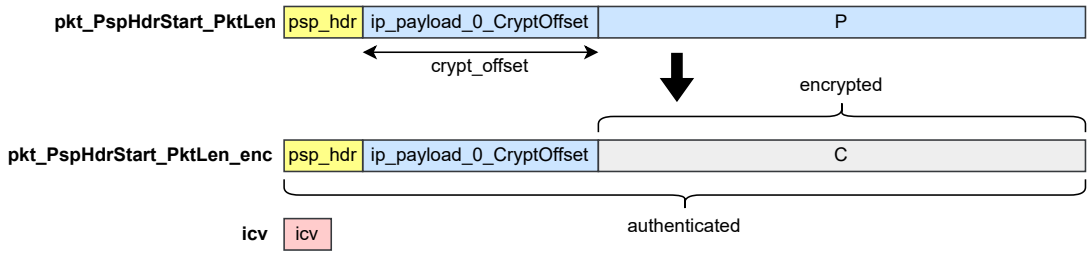

Figure 5.4: Authenticated encryption of data packet.

`iv` (64-bit) is first obtained from the IV generator block at line 1. Lines 3 and 4 extracts the `spi` (32-bit) from the PSP header, and concatenates it with `iv` at line 5 to form `gcm_iv` (96-bit) for AES GCM. The IP payload is split into segments before and after the crypt offset, which are named `ip_payload_0_CryptOffset` at line 8 and plaintext `P` at line 6, respectively. Line 7 inserts the `iv` into the PSP header so that the receiver has full information to decrypt the packet. Line 9 creates the additional authenticated data for AES GCM by concatenating the PSP header and `ip_payload_0_CryptOffset`. With all the input data prepared, lines 11-12 then initiates the call to the GCM library to perform authenticated encryption, using the same interface that we have previously described in Chapter 4. Line 13 then receives back the ciphertext `C` and authentication tag `icv`. Finally, line 15 combines the additional authenticated data with the ciphertext.

```
1  new iv: iv_t;
2
3  let (psp_hdr_no_iv:psp_hdr_t, =ip_payload)=pkt_PspHdrStart_PktLen in
4  let psp_header_partial(=crypt_offset, version, spi)=psp_hdr_no_iv in
5  let gcm_iv = (spi, iv) in
6  let P = ip_payload_after_CryptOffset(ip_payload, crypt_offset) in
7  let psp_hdr = psp_header(crypt_offset, version, spi, iv) in
8  let ip_payload_0_CryptOffset = ip_payload_before_CryptOffset(
       ip_payload, crypt_offset) in
9  let A = (psp_hdr, ip_payload_0_CryptOffset) in
10
11 new c_return: channel;
12 out(c_gcm_enc, (c_return, encryption_key, gcm_iv, P, P_blocks, A,
       A_blocks));
13 in(c_return, (C: bitstring, icv: bitstring));
14
15 let pkt_PspHdrStart_PktLen_enc = (psp_hdr, ip_payload_0_CryptOffset,
       C) in ...
```

Listing 5.1: IV generator and AES GCM encryption code.

5.3.4 Merge

The merge block simply pieces together the cleartext portion from the splitter block and the output from the AES GCM block to form the final PSP packet for transmission. Figure 5.5 shows how the individual packets are merged together by the merge block. We have implemented the merge block as a data constructor and our code is further explained in Appendix B.

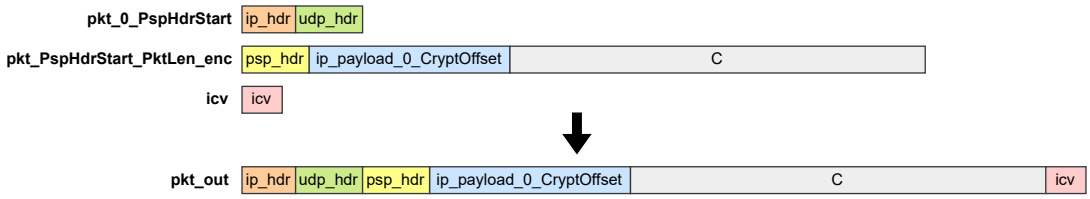

Figure 5.5: Merging of data packets into a PSP packet.

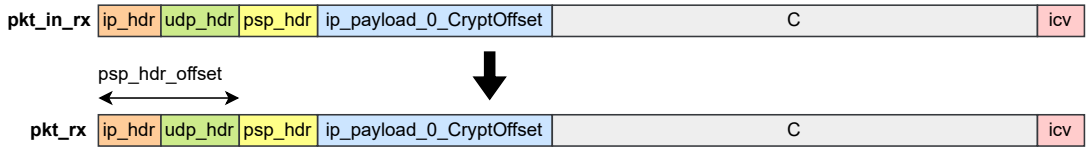

Figure 5.6: Parsing of PSP packet by ingress classifier.

5.4 NIC Receive

The receiver only possesses the master keys and does not need to store any decryption keys. This process uses the CMAC library for decryption key derivation, and the GCM library for authenticated decryption. Note in Figure 5.1 that this process contains a feedback loop that allows it to reuse the same master keys for messages within the same connection. We will refer to the block diagram of Figure 2.2 in Chapter 2 and describe our implementation of each component in this section. Some of the implementation code has been moved to Section B.2 of Appendix B, where they will be further explained.

5.4.1 Ingress classifier

The ingress classifier identifies the incoming packet as a PSP packet and communicates the PSP header offset to the splitter block. In our implementation, the identification of a PSP packet is implicitly achieved by blocking the process if it fails to parse a packet that has an incorrect format. Figure 5.6 shows how the incoming packet is handled, which in this case simply outputs the packet if the format is correct. We have implemented the ingress classifier as a destructor and our code is further explained in Appendix B.

5.4.2 Splitter

The splitter in the receiver process breaks up the incoming packet into six portions:

1. Cleartext portion of packet up to crypt offset, to be sent to the merge block.
2. Crypt offset from PSP header, to be sent to the AES GCM block.

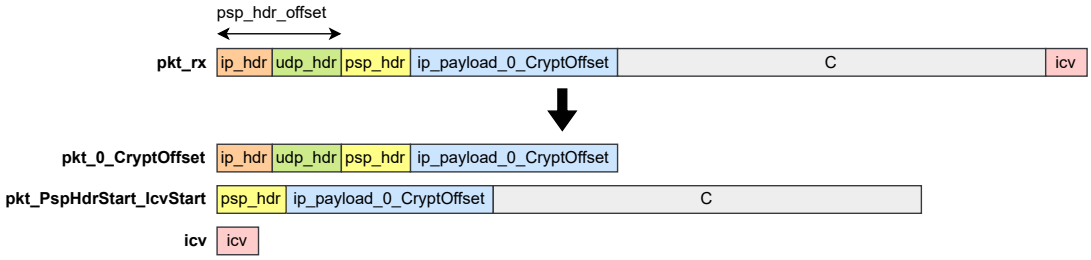

Figure 5.7: Division of PSP packet by splitter.

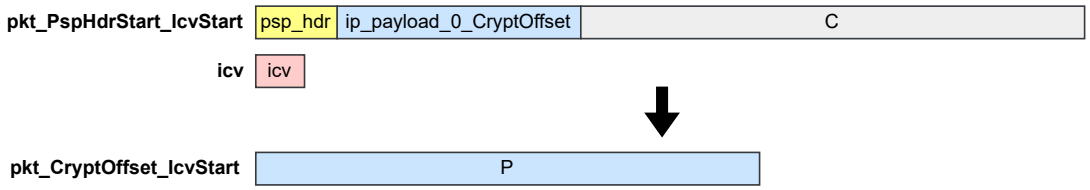

Figure 5.8: Authenticated decryption of PSP packet.

3. Portion from PSP header to before the ICV, to be sent to the AES GCM block.
4. SPI from PSP header, to be sent to the decryption key derivation block.
5. Most significant bit of the SPI, to be used for active master key selection.
6. ICV, for checking authenticity of received data.

We have implemented the splitter as a destructor and our code is further explained in Appendix B. Figure 5.7 illustrates how our implementation splits up the PSP packet.

5.4.3 Decryption key derivation, AES GCM (decrypt) and ICV check

Figure 5.8 shows how our implementation for the decryption key derivation, AES GCM and ICV check in the receiver process transform the data packet. Our code is shown in Listing 5.2. Lines 1-2 extract the PSP `version` and `iv` from the PSP header. These are the two terms which we had added into the receiver block diagram of Figure 2.2 after checking the C reference implementation. Lines 3-4 prepare the GCM IV (`gcm_iv`) and the additional authenticated data `A`. Lines 6-7 select the active master key using the most significant bit of the `spi`. Here, we show the case where master key 0 is the active key. Line 8 selects the AES algorithm using the `version`. Here, we show the case where 128-bit AES-GCM is selected. Lines 9-11 derives the decryption key. It interfaces with our CMAC library using the active master key as the CMAC key, with inputs being the `spi` and three other constants that depend on the `version`. Line 12 is a type converter that simply converts the type of the returned `mac` from `bitstring` to `key`. Finally, lines 14-16 interface with our GCM library and recovers plaintext

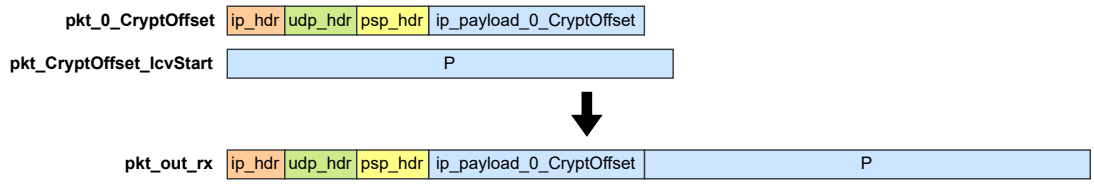

Figure 5.9: Merging of cleartext portions back into IP packet.

pkt_CryptOffset_IcvStart only if authentication is successful. The checking of icv is performed within the GCM library instead of using a separate component.

```
1  let (psp_hdr: psp_hdr_t, ip_payload_0_CryptOffset: bitstring, C:
       bitstring) = pkt_PspHdrStart_IcvStart in
2  let psp_header(=crypt_offset, version, =spi, iv) = psp_hdr in
3  let gcm_iv = (spi, iv) in
4  let A = (psp_hdr, ip_payload_0_CryptOffset) in
5
6  if spi_msb = BIT_0 then (
7    let active_master_key = master_key_0 in
8    if version = AES_GCM_128 then (
9      new c_return_cmac: channel;
10     out(c_cmac, (c_return_cmac, active_master_key, (HEX_00000001,
       HEX_50763000, spi, HEX_00000080), 1));
11     in(c_return_cmac, mac: bitstring);
12     let decryption_key = bitstring_to_key(mac) in
13
14     new c_return_gcm: channel;
15     out(c_gcm_dec, (c_return_gcm, decryption_key, gcm_iv, C,
       C_blocks, A, A_blocks, icv));
16     in(c_return_gcm, pkt_CryptOffset_IcvStart: bitstring); ...
```

Listing 5.2: Decryption key derivation, AES GCM decryption and ICV check code.

5.4.4 Merge

The merge block in the receive process simply puts together the cleartext portions to form the recovered IP packet, completing the PSP protocol. Figure 5.9 shows how the portions are merged together by the merge block. We have implemented the merge block as a data constructor and our code is further explained in Appendix B.

Chapter 6

Evaluation of PSP model

In this chapter, we will use ProVerif to formally verify the security goals of PSP by defining queries that test them against an active adversary. All events for queries on reachability and correspondence are placed at different checkpoints within a single PSP program, so that all the queries can be executed together. Other than basic tests for confidentiality and authenticity of the payload, we also carry out tests on other properties which the PSP protocol should exhibit, such as uniqueness of connection parameters and ability to run concurrent connections. Finally, we demonstrate some known weaknesses in the PSP protocol and show possible solutions to resolve them.

The verification results show that the PSP model passes all security and concurrency tests that we have designed. They will be explained in the following sections. We were also able to demonstrate some known attacks and fixes by modelling operations at layer 4 and higher. Other than these attacks, we did not discover any other flaws. Table 6.1 provides a summary of the results. All tests were executed on the selby compute server using psp.pv with `P_blocks = C_blocks = A_blocks = 1`, except for the isolation tests which use psp_leak_master_key.pv and psp_leak_sa_key.pv. All results have been archived at psp_results, psp_leak_master_key_results and psp_leak_sa_key_results folders.

6.1 Confidentiality of payload

We ran 41 secrecy queries to test the secrecy of all terms that appear in our PSP program. This includes `P`, the portion of the IP packet payload after the crypt offset. Queries on terms which are supposed to remain secret returned true, indicating that the main objective of confidentiality is indeed achieved in our symbolic model of PSP. The full list of secrecy queries and their results are documented in Section C.1 of Appendix C.

Security tests	No. of queries	Execution time
Confidentiality of payload	41	7 min 29 sec
Authenticity of payload	8	28 sec
Uniqueness of connection parameters	4	31 sec
Isolation between connections	24	16 min 53 sec
Impact of compromised devices	Same as the first two tests	
Replay attack and defence	12	2 min 59 sec
Bad-data-injection attack and defence	16	31 sec
Concurrency	8	4 min 17 sec

Table 6.1: Summary of verification results.

6.2 Authenticity of payload

To test the authenticity of the received payload, we first add events at the end of each process in the PSP model. When these events are queried individually, they would return true only if they can be reached by at least one execution of the protocol. Listing 6.1 contains two sets of queries that test the authenticity of the received payload. For instance, line 2 shows a query on an event named `psp_data_rx_0_128` located at the end of the receive process when master key 0 is active and the version is 128-bit AES. This event contains arguments `(mk0, mk1, k, iv, p, c, a, icv)` which are terms used within the receive process representing (master key 0, master key 1, decryption key, IV, recovered plaintext, ciphertext, additional authenticated data, authentication tag). A true result for this query means that this event has been reached. Line 3 contains a nested correspondence which not only allows us to test the authenticity of payload, but also allows us to test for the order of events. This query is true if and only if, for all executions of the protocol, if the event `psp_data_rx_0_128` has been reached, then the event `end_master_key` has been executed before `end_sa_0_128`, which has been executed before `psp_data_tx`, which has been executed before `psp_data_rx_0_128`.

```
1 query mk0: mkey_t, mk1: mkey_t, sa: sa_t, k: skey_t, iv: bitstring,
     p: bitstring, c: bitstring, a: bitstring, icv: bitstring;
2   event(psp_data_rx_0_128(mk0, mk1, k, iv, p, c, a, icv));
3   event(psp_data_rx_0_128(mk0, mk1, k, iv, p, c, a, icv)) ==> (event
      (psp_data_tx(sa, k, iv, p, c, a, icv)) ==> (event(end_sa_0_128(
      mk0, mk1, sa)) ==> event(end_master_key(mk0, mk1))));
4   event(psp_data_rx_0_256(mk0, mk1, k, iv, p, c, a, icv));
```

```
5  event(psp_data_rx_0_256(mk0, mk1, k, iv, p, c, a, icv)) ==> (event
     (psp_data_tx(sa, k, iv, p, c, a, icv)) ==> (event(end_sa_0_256(
     mk0, mk1, sa)) ==> event(end_master_key(mk0, mk1)))); ...
```

Listing 6.1: Authenticity queries.

We can represent these events using Figure 6.1, which is a simplified version of our PSP model from Figure 5.1. Events A, B, C, D represent `end_master_key`, `end_sa_0_128`, `psp_data_tx`, `psp_data_rx_0_128` with only main arguments included. Notice the terms `mk`, `sa` and `msg` appear in more than one event and link them together.

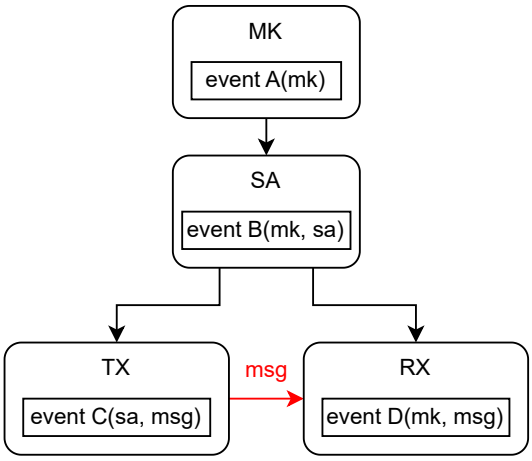

Figure 6.1: Events used in authenticity query.

When the authenticity queries are executed, we obtain true results for all of them. This means that the keys and messages are authentic and came from honest parties. Lines 4-5 are similar queries for the 256-bit version. We have omitted the queries corresponding to active master key 1, but they are similar to the ones listed here.

6.3 Uniqueness of connection parameters

Each run of the protocol should use a distinct set of master keys, encryption key and IV, because PSP guarantees that connections on the same transmitter or receiver are unique. We prove that for any two sets of connection parameters generated by PSP, if they match, then they must be from the same connection. Listing 6.2 shows our queries for the 128-bit version, where lines 2-3 are queries associated with active master key 0 and lines 4-5 are queries associated with active master key 1. Line 2 tests the reachability of two concurrent `psp_data_rx_0_128` events with the same master keys, encryption keys

and IVs, but potentially different values for the rest of the parameters consisting of the plaintext, ciphertext, additional authenticated data and authentication tag. Line 3 states that if the events on line 2 are reachable, then the rest of the parameters have the same values for these two events. We obtained a true result when we executed these queries. Since we have modelled different connections to initialise with different plaintexts and additional authenticated data, this result implies that these two events must belong to the same connection, hence proving that the connection parameters are indeed unique.

```
1 query mk0: mkey_t, mk1: mkey_t, k: skey_t, iv: bitstring, p:
     bitstring, c: bitstring, a: bitstring, icv: bitstring, p':
     bitstring, c': bitstring, a': bitstring, icv': bitstring;
2   event(psp_data_rx_0_128(mk0, mk1, k, iv, p, c, a, icv)) && event(
     psp_data_rx_0_128(mk0, mk1, k, iv, p', c', a', icv'));
3   event(psp_data_rx_0_128(mk0, mk1, k, iv, p, c, a, icv)) && event(
     psp_data_rx_0_128(mk0, mk1, k, iv, p', c', a', icv')) ==> (p, c,
     a, icv) = (p', c', a', icv');
4   event(psp_data_rx_1_128(mk0, mk1, k, iv, p, c, a, icv)) && event(
     psp_data_rx_1_128(mk0, mk1, k, iv, p', c', a', icv'));
5   event(psp_data_rx_1_128(mk0, mk1, k, iv, p, c, a, icv)) && event(
     psp_data_rx_1_128(mk0, mk1, k, iv, p', c', a', icv')) ==> (p, c,
     a, icv) = (p', c', a', icv').
```

Listing 6.2: Queries to test uniqueness of connection parameters.

6.4 Isolation between connections

Another approach to verify that the connection parameters are unique is to intentionally leak some key parameters from one connection and see whether this affects the secrecy and authenticity of the payload from another connection (it should not). We performed two kinds of checks for the isolation property using psp_leak_master_key.pv and psp_leak_sa_key.pv programs. In the first program, we set up two connections using different master keys and then expose the master keys in one of them. This is illustrated by Figure 6.2, where terms in red have been leaked. In the second program, we set up two connections using the same master keys and then expose the SA key in one of them. This is illustrated by Figure 6.3. The queries used to test for isolation are shown in Section C.2 of Appendix C. Our results show that the secrecy and authenticity of the payload is preserved in the connection that has not been compromised, proving that the isolation property is indeed satisfied by our model.

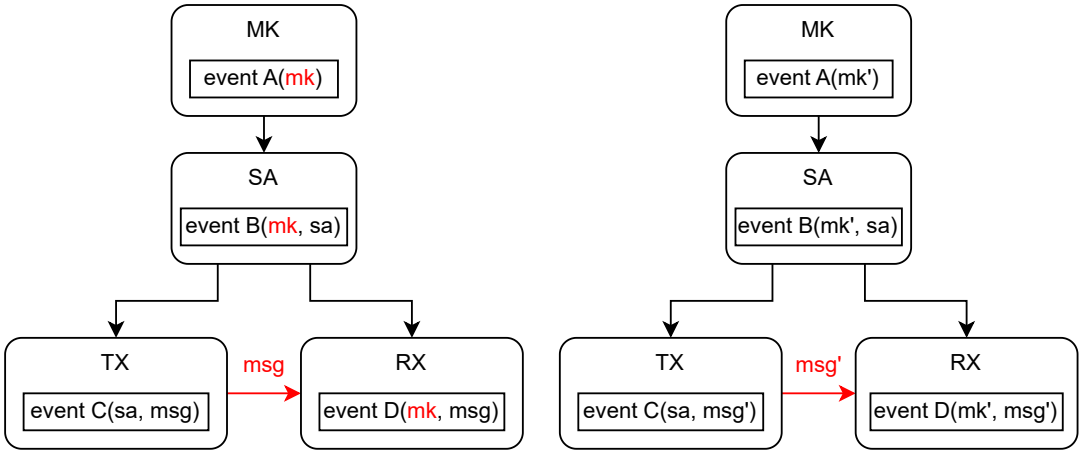

Figure 6.2: Leaking master keys in left connection does not affect right connection.

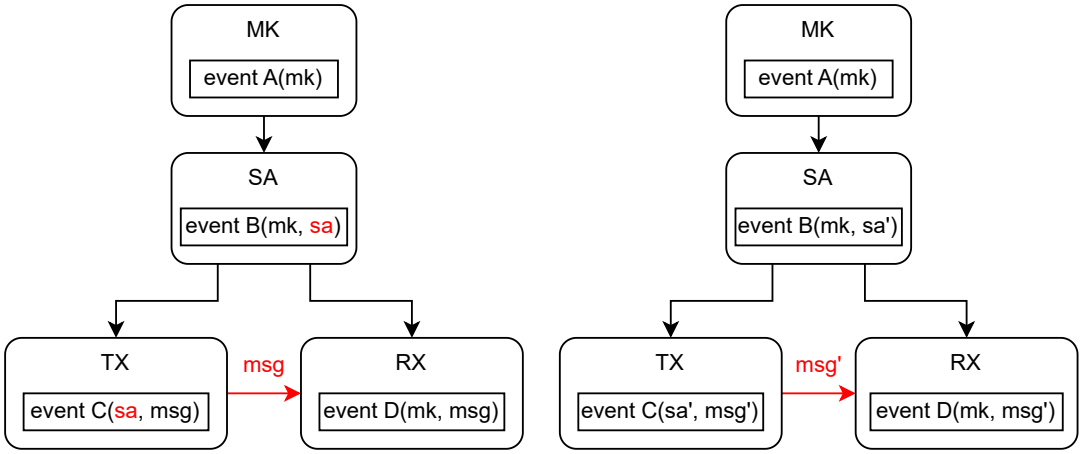

Figure 6.3: Leaking SA key in left connection does not affect right connection.

6.5 Impact of compromised devices

We show that under an alternative threat model where some component in the transmitter or receiver has been compromised by the adversary, the transmitted data would no longer be confidential or authentic. We do this by disclosing each secret term in our PSP model to the adversary (i.e. sending it to a public channel) and observing that leaking any single one of them would compromise the secrecy and authenticity of the plaintext. This result is expected since any leaked term would allow the adversary to derive the SA key and decrypt or forge new messages. The queries are the same as those in Sections 6.1 and 6.2, with the results now being false. This study also implies that the set of terms that are currently kept secret is already an optimal set with the minimum number of secret terms.

6.6 Replay attack and defence

The PSP specification states that PSP does not provide replay protection and assumes that replay protection will be provided by the layer 4 protocol. We first show that replay attack can happen by running a set of queries to test for injective correspondence in Listing 6.3 (only showing one of four cases). This query is true if for each occurrence of the event `psp_data_rx_0_128`, there is a distinct earlier occurrence of the event `psp_data_tx`. Our execution produces a false result because the adversary was able to replay the message that was transmitted to the receiver.

```
1 query mk0: mkey_t, mk1: mkey_t, sa: sa_t, k: skey_t, iv: bitstring,
     p: bitstring, c: bitstring, a: bitstring, icv: bitstring;
2   inj-event(psp_data_rx_0_128(mk0, mk1, k, iv, p, c, a, icv)) ==>
     inj-event(psp_data_tx(sa, k, iv, p, c, a, icv)); ...
```

Listing 6.3: Queries to test replay attack.

We then enhanced the PSP model by modelling replay protection at layer 4. This is used to demonstrate the possible fix and is not meant to be part of the PSP protocol. Figure 6.4 illustrates the additional operations needed compared to Figure 6.1. We first set up a TCP connection between transmitter and receiver so that the transmitter can obtain an expected packet sequence number from the receiver. The transmitter then includes this sequence number into the layer 4 header when sending the PSP packet over to the receiver. The receiver would check the sequence number in the layer 4 header to ensure that this is a fresh packet. An adversary who tries to replay the packet would be unsuccessful since the sequence number would have already been used. Events A, B, C and D have been described in Section 6.2. We added event E in the receive process after the sequence number check, which refers to `l4_data_rx_0_128` shown in Listing 6.4 to test for replay protection at layer 4 (only showing one case). This new query has a true result, indicating that replay protection has indeed been established at layer 4.

```
1 query spi: bitstring, mk0: mkey_t, mk1: mkey_t, sa: sa_t, k: skey_t,
     iv: bitstring, p: bitstring, c: bitstring, a: bitstring, icv:
     bitstring;
2   event(l4_data_rx_0_128(mk0, mk1, k, iv, p, c, a, icv));
3   inj-event(l4_data_rx_0_128(mk0, mk1, k, iv, p, c, a, icv)) ==> inj
     -event(psp_data_tx(sa, k, iv, p, c, a, icv)); ...
```

Listing 6.4: Queries to test replay protection.

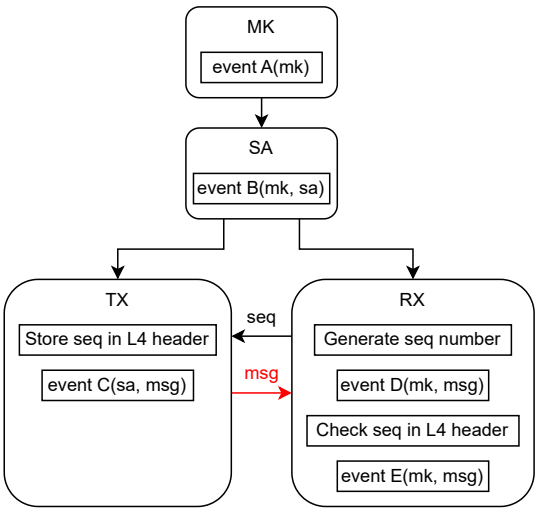

Figure 6.4: Additional operations to show replay protection at layer 4.

6.7 Bad-data-injection attack and defence

Another attack described in PSP specification is known as the bad-data-injection attack. This happens when a transmit process attempts to send packets to the receive process, but by injecting it into another connection instead of using its own connection. This transmitter is a malicious actor with a valid SA that is different from the SA associated with the other connection. Both SAs are valid as they have been derived from the same master key on the receive process. Upon receiving the forged packet on the other connection, the receive process will decrypt and accept the forged packet since it uses a valid SA and there is no indication that the packet does not belong to the connection.

We first show that this attack can occur in our PSP model by adding some new events that reveal the expected SPI values for each connection. Since different connections will have different SPI values, enriching our events with this term allows us to easily distinguish one connection from another. This is modelled in Figure 6.5, where events F and G have been added. Referring to Listing 6.5, event F corresponds to `psp_data_with_spi_tx`, while event G corresponds to `l4_data_with_spi_rx_0_128` (showing only one case). The result from executing this query is false, which indicates that a forged packet with a different SPI from the designated transmitter could be accepted by the receive process.

```
1 query spi: bitstring, mk0: mkey_t, mk1: mkey_t, sa: sa_t, k: skey_t,
      iv: bitstring, p: bitstring, c: bitstring, a: bitstring, icv:
    bitstring;
2   event(l4_data_with_spi_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
```

```
  ));
3 event(l4_data_with_spi_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
  )) ==> event(psp_data_with_spi_tx(spi, sa, k, iv, p, c, a, icv));
   ...
```

Listing 6.5: Queries to test bad-data-injection attack.

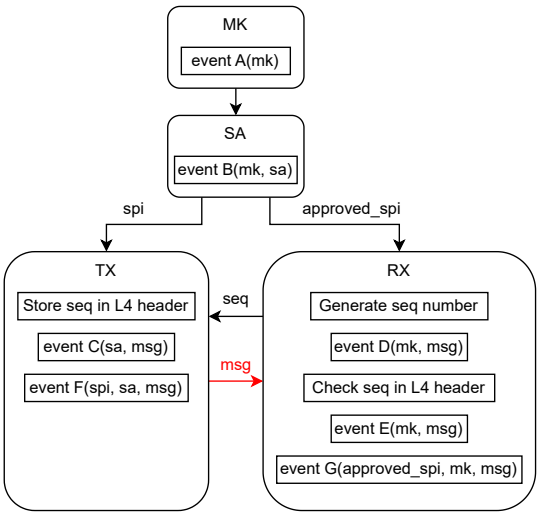

Figure 6.5: Additional events to show bad-data-injection attack.

The solution to defend against this attack is for the receive process to provide the SPI information to upper-layer software. The upper-layer software keeps track of the valid SPI for this connection, which allows the packet to be dropped if the SPI is not on the approved list. This is modelled by Figure 6.6, where we have added the SPI check followed by an additional event H into our PSP program. Event H corresponds to `upper_layer_data_rx_0_128` in Listing 6.6 (showing only one case) where we check for correspondence between event H and event F. This query is true, indicating that only packets with approved SPI would be accepted.

```
1 query spi: bitstring, mk0: mkey_t, mk1: mkey_t, sa: sa_t, k: skey_t,
     iv: bitstring, p: bitstring, c: bitstring, a: bitstring, icv:
   bitstring;
2  event(upper_layer_data_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
   ));
3  event(upper_layer_data_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
   )) ==> event(psp_data_with_spi_tx(spi, sa, k, iv, p, c, a, icv));
    ...
```

Listing 6.6: Queries to test bad-data-injection defence.

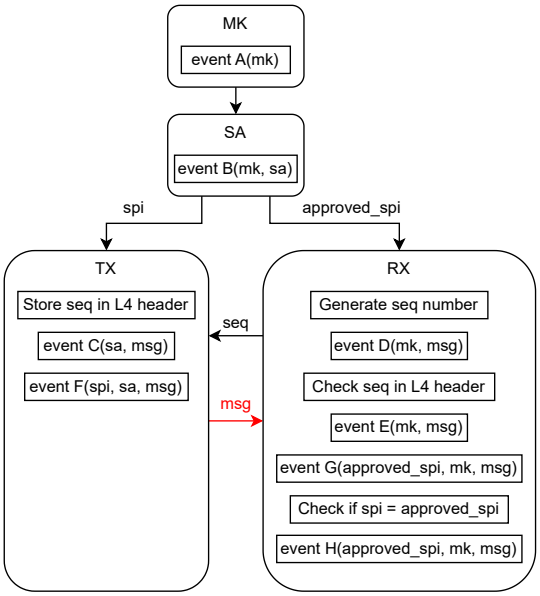

Figure 6.6: Additional operations to show bad-data-injection defence.

6.8 Concurrency

To verify that our final PSP program is rich enough to model different scenarios involving multiple transactions and connections, we designed a series of reachability queries to simulate different scenarios involving concurrency. Executing these queries would then output trace graphs which confirm that these scenarios can be modelled by our program. The scenarios that we have tested are:

1. Consecutive transmissions within the same connection
2. Concurrent connections using same master keys but different SA keys
3. Concurrent connections using master keys 0 and 1 on same receiver
4. Concurrent connections using different master keys on different receivers

The queries to simulate these scenarios and their simplified output trace graphs can be found in Section C.3 of Appendix C.

Chapter 7

Conclusion

In this project, we developed a symbolic model of the PSP protocol using ProVerif and verified all the security guarantees that PSP claims to offer to its users. The primary security tests verify confidentiality and authenticity of data-in-transit, while secondary ones check for uniqueness of connection parameters and that the connections are isolated from one another. These proofs are valid under certain conditions that were imposed in this study due to resource or tool constraints. For instance, messages need to be below some block number to limit the analysis time, and not all XOR properties have been modelled due to ProVerif limitations. Taking these constraints into consideration and excluding known weaknesses of PSP, our study did not discover any other flaws in the symbolic model of PSP.

The PSP model made use of the CMAC and GCM models which we had also developed. We separately verified the security goals of these algorithms and showed that the models satisfy properties such as EUF-CMA and IND-CCA2. Since CMAC and GCM are popular algorithms used by cryptographic protocols, we have packaged them into ProVerif library files to facilitate deployment.

Another unique feature of our work is that we catered for input messages with arbitrary number of blocks and explicitly modelled the iterations within the CMAC and GCM algorithms. This made our PSP model more robust as it can support data packets with arbitrary sizes. Furthermore, the modelling of multiple message blocks enabled the discovery of subtle attacks, such as the length extension attack, when using an incorrect implementation of the CMAC or GCM algorithm.

We then reviewed potential problems that could arise when using the PSP protocol. One of them involved an alternative threat model where devices have been compromised. We showed that under this threat model, the security of the PSP protocol would be

completely eroded. We also used our model to demonstrate known weaknesses of PSP such as replay attacks and bad data injection. Fixes were then added into the program to show that these attacks can be mitigated as long as the developer handles them correctly.

Finally, we conducted a series of concurrency tests and reviewed their trace graphs to confirm that our PSP program is indeed capable of launching multiple transactions and connections at the same time. These results showed that our model is versatile and able to produce realistic scenarios.

7.1 Future work

An area for future work is to extend the verification of PSP model to the computational model. As previously introduced in Chapter 2, the computational model is closer to the actual execution of protocols and allows for a larger class of attacks to be analysed. This is partly because the messages are now modelled as actual bitstrings that can be manipulated by the adversary. One limitation of the symbolic model arises from the issue of computational soundness. While an attack in the symbolic model directly results in an attack in the computational model, security in the symbolic model does not necessarily imply the same in the computational model, unless computational soundness is proven in the symbolic model of the protocol.

There are two possible approaches to reconcile the symbolic and computational models of PSP. The first approach is to derive a computational soundness theorem applicable to PSP model so that security results from the symbolic model would apply to the computational model. For instance, Abadi and Rogaway had introduced a soundness theorem showing that equivalence in formal setting implies indistinguishability in computational setting for symmetric encryption [32]. Further work from Micciancio and Warinschi showed that if an asymmetric encryption scheme is proven to be IND-CCA2 secure in the symbolic model, then its authentication properties are also satisfied in the computational model [33]. The second approach is to directly derive proofs of security for PSP in computational model without relying on the symbolic model, but with the help of tools designed for such purposes [16]. CryptoVerif is one such tool that can be used to prove secrecy, correspondences, and indistinguishability in the computational setting [34]. Another tool is EasyCrypt, which can also be used to verify the security of cryptographic constructions in the computational model [8]. Proving the security of PSP in the computational setting would complement the results from our project and help achieve a more holistic security proof for PSP.

Bibliography

[1] WhatsApp, LLC, "Whatsapp encryption overview technical white paper." https://scontent-lhr8-2.xx.fbcdn.net/v/t39.8562-6/383236184_722587863039320_5040651063228680393_n.pdf?_nc_cat=101&ccb=1-7&_nc_sid=b8d81d&_nc_ohc=F7fnY0N0DC0Q7kNvgHvEyaG&_nc_ht=scontent-lhr8-2.xx&oh=00_AYB5g7Q5EQHyjT4Y3OurjwUTBI4JSwxTqcN7JirhGC3Xvg&oe=669EDB44, 2023. Accessed 18 Jul 2024.

[2] Google Cloud, "Encryption in transit." https://cloud.google.com/docs/security/encryption-in-transit#network_encryption_using_psp, 2022. Accessed 17 Jul 2024.

[3] A. Vahdat and S. H. Yeganeh, "Announcing psp's cryptographic hardware offload at scale is now open source." https://cloud.google.com/blog/products/identity-security/announcing-psp-security-protocol-is-now-open-source, 2022. Accessed 17 Jul 2024.

[4] Google LLC, "Psp architecture specification." https://github.com/google/psp/blob/main/doc/PSP_Arch_Spec.pdf, 2022. Accessed 18 Jul 2024.

[5] V. Cortier and S. Kremer, eds., *Formal Models and Techniques for Analyzing Security Protocols*, vol. 5 of *Cryptology and Information Security Series*. IOS Press, 2011.

[6] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[7] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Found. Trends Priv. Secur.*, vol. 1, no. 1-2, pp. 1–135, 2016.

[8] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "Sok: Computer-aided cryptography," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pp. 777–795, IEEE, 2021.

[9] B. Blanchet and V. Cheval, "Proverif: Cryptographic protocol verifier in the formal model." `https://bblanche.gitlabpages.inria.fr/proverif/`. Accessed 18 Jul 2024.

[10] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pp. 483–502, IEEE Computer Society, 2017.

[11] K. Bhargavan, V. Cheval, and C. A. Wood, "A symbolic analysis of privacy for TLS 1.3 with encrypted client hello," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022* (H. Yin, A. Stavrou, C. Cremers, and E. Shi, eds.), pp. 365–379, ACM, 2022.

[12] Google LLC, "Github - google/psp." `https://github.com/google/psp`. Accessed 21 Jul 2024.

[13] M. J. Dworkin, "Recommendation for block cipher modes of operation: The cmac mode for authentication," *Special Publication (NIST SP), National Institute of Standards and Technology*, 2016.

[14] M. J. Dworkin, "Recommendation for block cipher modes of operation: Galois/-counter mode (gcm) and gmac," *Special Publication (NIST SP), National Institute of Standards and Technology*, 2007.

[15] T. Mieno, H. Okazaki, K. Arai, and Y. Futa, "How to formalize loop iterations in cryptographic protocols using proverif," *IEEE Access*, 2024.

[16] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*

(P. Degano and J. D. Guttman, eds.), vol. 7215 of *Lecture Notes in Computer Science*, pp. 3–29, Springer, 2012.

[17] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[18] C. Meadows, "The NRL protocol analyzer: An overview," *J. Log. Program.*, vol. 26, no. 2, pp. 113–131, 1996.

[19] S. Escobar, C. Meadows, and J. Meseguer, "A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties," *Theor. Comput. Sci.*, vol. 367, no. 1-2, pp. 162–202, 2006.

[20] B. Schmidt, S. Meier, C. Cremers, and D. A. Basin, "Automated analysis of diffie-hellman protocols and advanced security properties," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012* (S. Chong, ed.), pp. 78–94, IEEE Computer Society, 2012.

[21] C. J. F. Cremers, "Unbounded verification, falsification, and characterization of security protocols by pattern refinement," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008* (P. Ning, P. F. Syverson, and S. Jha, eds.), pp. 119–128, ACM, 2008.

[22] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, "Verified cryptographic implementations for TLS," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 3:1–3:32, 2012.

[23] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, "Proverif 2.05: automatic cryptographic protocol verifier, user manual and tutorial." `https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf`, 2023. Accessed 20 Jul 2024.

[24] K. Arai, D. Watanabe, and H. Sakurada, "Formal verification of tls 1.3 full handshake protocol using proverif," tech. rep., Technical report, Cryptographic protocol Evaluation toward Long-Lived Outstanding Security, 2016.

[25] K. Arai and S. Matsuo, "Formal verification of tls 1.3 full handshake protocol using proverif (draft-11). ietf tls mailing list (2016)."

[26] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pp. 435–450, IEEE, 2017.

[27] J. Black and P. Rogaway, "CBC macs for arbitrary-length messages: The three-key constructions," *J. Cryptol.*, vol. 18, no. 2, pp. 111–131, 2005.

[28] T. Iwata and K. Kurosawa, "OMAC: one-key CBC MAC," in *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers* (T. Johansson, ed.), vol. 2887 of *Lecture Notes in Computer Science*, pp. 129–153, Springer, 2003.

[29] D. A. McGrew and J. Viega, "The security and performance of the galois/counter mode (GCM) of operation," in *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings* (A. Canteaut and K. Viswanathan, eds.), vol. 3348 of *Lecture Notes in Computer Science*, pp. 343–355, Springer, 2004.

[30] D. McGrew and J. Viega, "The galois/counter mode of operation (gcm)," *submission to NIST Modes of Operation Process*, vol. 20, pp. 0278–0070, 2004.

[31] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[32] M. Abadi and P. Rogaway, "Reconciling two views of cryptography (the computational soundness of formal encryption)," *J. Cryptol.*, vol. 15, no. 2, pp. 103–127, 2002.

[33] D. Micciancio and B. Warinschi, "Soundness of formal encryption in the presence of active adversaries," in *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings* (M. Naor, ed.), vol. 2951 of *Lecture Notes in Computer Science*, pp. 133–151, Springer, 2004.

[34] B. Blanchet, A. Fromherz, C. Jacomme, and B. Lipp, "Cryptoverif: Cryptographic protocol verifier in the computational model." `https://bblanche.gitlabpages.inria.fr/CryptoVerif/`. Accessed 31 Jul 2024.

Appendix A

ProVerif syntax

A.1 Example program

We provide an explanation of ProVerif syntax with the aid of a simple program shown in Listing A.1. This program can be executed using the command: "`proverif example.pv`" on a system with ProVerif 2.05 installed. This command assumes that the current working directory contains the ProVerif binary and the program file example.pv that holds the code in Listing A.1.

```
1  (* ProVerif example code *)
2
3  type key.
4
5  free c: channel.
6  free k: key [private].
7
8  fun enc(key, bitstring): bitstring.
9  reduc forall k: key, m: bitstring; dec(k, enc(k, m)) = m.
10
11 event end_sender(bitstring).
12 event end_receiver(bitstring).
13
14 query attacker(k).
15 query secret plaintext.
16 query m: bitstring; event(end_receiver(m)) ==> event(end_sender(m)).
17
18 let sender =
19   new plaintext: bitstring;
20   let ciphertext = enc(k, plaintext) in
```

```
21    event end_sender(plaintext);
22    out(c, ciphertext).
23
24  let receiver =
25    in(c, received_ciphertext: bitstring);
26    let recovered_plaintext = dec(k, received_ciphertext) in
27    event end_receiver(recovered_plaintext).
28
29  process
30    ( (!sender) | (!receiver) )
31
32  (* EXPECTPV
33  RESULT not attacker(k[]) is true.
34  RESULT secret plaintext is true.
35  RESULT event(end_receiver(m)) ==> event(end_sender(m)) is true.
36  0.021s (elapsed: user + system + other processes)
37  END *)
```

Listing A.1: Example program.

Lines 29-30 show the main process that runs when the program is executed. This example launches an unbounded number of sessions (due to `!`) of the `sender` and `receiver` processes, which represent participants of the protocol running in parallel (due to `|`). Lines 18-22 define the actions of the `sender` process, who creates a new `plaintext` of type `bitstring`, encrypts it using the `enc` function, stores it as the `ciphertext` and sends it out to a public channel which is accessible by an attacker. Line 24-27 define the actions of the `receiver` process, who accepts the `ciphertext` from the public channel, attempts to decrypt it using the same key, and stores the result as `recovered_plaintext` if decryption is successful. Lines 11-12 declares events that are placed at lines 21 and 27. They mark important checkpoints reached by the protocol but do not affect its behaviour.

We next review the declarations at the top of the program. Line 1 shows a comment enclosed between `(*` and `*)`. Line 3 declares a user-defined type: `key`. Types are solely used for type-checking the program and ProVerif ignores types during verification. Line 5 declares a free name `c` of `channel` type, which is a built-in type. This channel will be used for public communication of messages. Line 6 declares a free name `k` of `key` type, and uses the keyword `[private]` to exclude `k` from the attacker's knowledge. Line 8 declares a constructor `enc` that abstracts the encryption function, while line 9 declares a destructor `dec` that recovers message `m` from an encrypted blob only if the decryption

key matches the encryption key.

Queries can be used to test the security properties of a protocol. Lines 14-15 (reproduced below) are confidentiality tests, which are associated with reachability properties in ProVerif. These queries check whether key `k` and the `plaintext` are available to the attacker.

```
query attacker(k).
query secret plaintext.
```

Line 16 (reproduced below) is a query for message authenticity, which is associated with correspondence properties in ProVerif. This query is true if and only if, for all executions of the protocol, if the event `end_receiver` with argument `m` has been executed, then the event `end_sender` with argument `m` has also been executed before.

```
query m: bitstring; event(end_receiver(m)) ==> event(end_sender(m)).
```

Executing the program outputs true for all three queries, which indicate that the secrecy and authenticity properties of the protocol are satisfied.

```
Verification summary:
Query not attacker(k[]) is true.
Query secret plaintext is true.
Query event(end_receiver(m)) ==> event(end_sender(m)) is true.
```

A.2 List of syntax used in project

We list the syntax that are used in this project. These are extracted from the official ProVerif user manual at [23].

$M, N ::=$	terms
x, a, c	variable, free name, or constant
$0, 1, \ldots$	natural numbers
$(M_1, \ldots, M_n)$	tuple
$h(M_1, \ldots, M_n)$	constructor/destructor application
$M + i$	addition, $i \in \mathbb{N}$
$M - i$	subtraction, $i \in \mathbb{N}$
$M > N$	greater
$M = N$	term equality
$M <> N$	term disequality
M && M	conjunction

P, Q ::=	processes
$P \mid Q$	parallel composition
$!P$	replication
new $n : t; P$	name restriction
$\text{in}(M, x : t); P$	message input
$\text{out}(M, N); P$	message output
if M then P else Q	conditional
let $x = M$ in P	term evaluation
$R(M_1, \ldots, M_k)$	macro usage
event $e(M_1, \ldots, M_n); P$	event
phase $n; P$	phase
T ::=	patterns
$x : t$	typed variable
x	variable without explicit type
$(T_1, \ldots, T_n)$	tuple
$=M$	equality test
q ::=	query
F_1 && $\ldots$ && F_n	reachability (F ::= fact)
F_1 && $\ldots$ && F_n ==> H	correspondence (H ::= hypothesis)
A, B ::=	biterm
choice$[A, B]$	choice

Appendix B

Supplementary code in NIC Transmit and Receive processes

B.1 NIC Transmit

We show and explain the code used for some of the components in the NIC Transmit process. These components perform non-cryptographic operations and were left out of Section 5.3 due to space constraints.

B.1.1 Egress classifier

```
1  fun security_association(bitstring, skey_t, offset_t, version_t):
       sa_t [data].
2  fun udp_header(bitstring): udp_hdr_t.
3  fun psp_header_partial(offset_t, version_t, bitstring): psp_hdr_t [
       data].
4  fun psp_header_offset(ip_hdr_t, udp_hdr_t): offset_t.
5
6  reduc forall spi: bitstring, sa_key: skey_t, crypt_offset: offset_t,
        version: version_t, ip_hdr: ip_hdr_t, ip_payload: bitstring;
7    let sa = security_association(spi,sa_key,crypt_offset,version) in
8    let pkt_in = (ip_hdr, ip_payload) in
9    let udp_hdr = udp_header(ip_payload) in
10   let psp_hdr = psp_header_partial(crypt_offset, version, spi) in
11   let pkt = (ip_hdr, udp_hdr, psp_hdr, ip_payload) in
12   let psp_hdr_offset = psp_header_offset(ip_hdr, udp_hdr) in
13   egress_classifier(sa, pkt_in) = (sa_key, pkt, psp_hdr_offset).
14
```

```
15 let (encryption_key: skey_t, pkt: bitstring, psp_hdr_offset:
       offset_t) = egress_classifier(sa, pkt_in) in ...
```

Listing B.1: Egress classifier code.

Listing B.1 shows snippets from our implementation of the egress classifier and follows the description in Section 5.3.1. We defined relevant constructors that creates the SA, UDP header, PSP header and PSP header offset at lines 1-4. Some constructors have keyword `[data]` appended, which means the input terms can be recovered from the constructor output. We then create an `egress_classifier` destructor at lines 6-13, and used it in the NIC Transmit process. Line 15 shows what the instruction in the NIC Transmit process looks like. The egress classifier takes in a new IP packet `pkt_in` and outputs a transformed packet `pkt`, among other input and output terms.

B.1.2 Splitter

```
1  reduc forall sa_key: skey_t, crypt_offset: offset_t, version:
       version_t, spi: bitstring, ip_hdr: ip_hdr_t, udp_hdr: udp_hdr_t,
       ip_payload: bitstring;
2    let psp_hdr = psp_header_partial(crypt_offset, version, spi) in
3    let pkt = (ip_hdr, udp_hdr, psp_hdr, ip_payload) in
4    let psp_hdr_offset = psp_header_offset(ip_hdr, udp_hdr) in
5    let pkt_0_PspHdrStart = (ip_hdr, udp_hdr) in
6    let pkt_PspHdrStart_PktLen = (psp_hdr, ip_payload) in
7    splitter_tx(sa_key, pkt, psp_hdr_offset) = (pkt_0_PspHdrStart,
       sa_key, crypt_offset, pkt_PspHdrStart_PktLen).
8
9  let (pkt_0_PspHdrStart: bitstring, =encryption_key, crypt_offset:
       offset_t, pkt_PspHdrStart_PktLen: bitstring) = splitter_tx(
       encryption_key, pkt, psp_hdr_offset) in ...
```

Listing B.2: Splitter code in transmit process.

Listing B.2 shows snippets from our implementation of the splitter in the transmit process and follows the description in Section 5.3.2. We defined a `splitter_tx` destructor at lines 1-7, and used it in the main program at line 9. The splitter manipulates the input terms and reduces it into four other terms, with `pkt_0_PspHdrStart` representing the data segment from the start of packet to just before the PSP header, and `pkt_PspHdrStart_PktLen` representing the data segment from the PSP header to end of packet. The use of '=' in '`=encryption_key`' is a pattern matching syntax that

requires the `encryption_key` output from the splitter to match the encryption key that had been previously sent to NIC Transmit from the SA generation and distribution process.

B.1.3 Merge

```
1 fun merge_tx(bitstring, bitstring, bitstring): bitstring [data].
2
3 let pkt_out = merge_tx(pkt_0_PspHdrStart, pkt_PspHdrStart_PktLen_enc
     , icv) in
4 out(c_public, pkt_out)
```

Listing B.3: Merge code in transmit process.

Listing B.3 shows snippets from our implementation of the merge block in the transmit process and follows the description in Section 5.3.4. Line 1 declares the data constructor for the merge block, where the inputs can be recovered from the output. Line 3 merges the input data packets into the PSP packet `pkt_out`, before sending it to the receiver via the public channel at line 4.

B.2 NIC Receive

We show and explain the code used for some of the components in the NIC Receive process. These components perform non-cryptographic operations and were left out of Section 5.4 due to space constraints.

B.2.1 Ingress classifier

```
1 reduc forall ip_hdr: ip_hdr_t, udp_hdr: udp_hdr_t, psp_hdr:
     psp_hdr_t, ip_payload_0_CryptOffset: bitstring, ciphertext:
     bitstring, icv: bitstring;
2   let pkt_0_PspHdrStart = (ip_hdr, udp_hdr) in
3   let pkt_PspHdrStart_PktLen_enc = (psp_hdr,
     ip_payload_0_CryptOffset, ciphertext) in
4   let pkt_in = merge_tx(pkt_0_PspHdrStart,
     pkt_PspHdrStart_PktLen_enc, icv) in
5   let pkt = pkt_in in
6   let psp_hdr_offset = psp_header_offset(ip_hdr, udp_hdr) in
7   ingress_classifier(pkt_in) = (pkt, psp_hdr_offset).
```

```
8
9  let (pkt_rx: bitstring, psp_hdr_offset: offset_t) =
       ingress_classifier(pkt_in_rx) in ...
```

Listing B.4: Ingress classifier code.

Listing B.4 shows snippets from our implementation of the ingress classifier and follows the description in Section 5.4.1. Lines 1-7 declares a destructor for the ingress classifier, where it parses the incoming PSP packet and derives the PSP header offset. Line 9 can be found inside the NIC Receive process, where it calls the destructor using the incoming PSP packet `pkt_in_rx` and outputs `pkt_rx` and `psp_hdr_offset`.

B.2.2 Splitter

```
1  reduc forall ip_hdr: ip_hdr_t, udp_hdr: udp_hdr_t, spi_msb:
       bitstring, spi_31lsb: bitstring, crypt_offset: offset_t, version:
        version_t, iv: iv_t, ip_payload_0_CryptOffset: bitstring,
       ciphertext: bitstring, icv: bitstring;
2    let pkt_0_PspHdrStart = (ip_hdr, udp_hdr) in
3    let spi = (spi_msb, spi_31lsb) in
4    let psp_hdr = psp_header(crypt_offset, version, spi, iv) in
5    let pkt_PspHdrStart_IcvStart = (psp_hdr, ip_payload_0_CryptOffset,
       ciphertext) in
6    let pkt = merge_tx(pkt_0_PspHdrStart, pkt_PspHdrStart_IcvStart,
       icv) in
7    let psp_hdr_offset = psp_header_offset(ip_hdr, udp_hdr) in
8    let pkt_0_CryptOffset = (ip_hdr, udp_hdr, psp_hdr,
       ip_payload_0_CryptOffset) in
9    splitter_rx(pkt, psp_hdr_offset) = (pkt_0_CryptOffset,
       crypt_offset, pkt_PspHdrStart_IcvStart, spi, icv, spi_msb).
10
11 let (pkt_0_CryptOffset: bitstring, crypt_offset: offset_t,
       pkt_PspHdrStart_IcvStart: bitstring, spi: bitstring, icv:
       bitstring, spi_msb: bitstring) = splitter_rx(pkt_rx,
       psp_hdr_offset) in ...
```

Listing B.5: Splitter code in receive process.

Listing B.5 shows snippets from our implementation of the splitter in the receive process and follows the description in Section 5.4.2. We defined a `splitter_rx` destructor at lines 1-9, and used it in the main program at line 11. The splitter manipulates the input terms and reduces it into six other terms, with three of them representing

segments of the PSP packet. `pkt_0_CryptOffset` represents the data segment from the start of packet to the crypt offset, while `pkt_PspHdrStart_IcvStart` represents the data segment from the PSP header to just before the ICV. The `spi` and `crypt_offset` terms are extracted from the PSP header.

B.2.3 Merge

```
1 fun merge_rx(bitstring, bitstring): bitstring [data].
2
3 let pkt_out_rx = merge_rx(pkt_0_CryptOffset,
    pkt_CryptOffset_IcvStart) in ...
```

Listing B.6: Merge code in receive process.

Listing B.6 shows snippets from our implementation of the merge block in the receive process and follows the description in Section 5.4.4. Line 1 declares the data constructor for the merge block, where the inputs can be recovered from the output. Line 3 then merges the segments `pkt_0_CryptOffset` and `pkt_CryptOffset_IcvStart` into a single packet `pkt_out_rx`.

Appendix C

Supplementary queries in PSP model

C.1 Secrecy queries and results

We elaborate on the secrecy tests introduced in Section 6.1. Listing C.1 shows the complete list of 41 secrecy queries in our PSP program. They have been grouped according to their results as true (secret) or false (non-secret). The results are consistent because terms which are meant to be confidential have a true result, while terms which are public knowledge have a false result.

```
(* Result: true *)
query secret master_key_0.
query secret master_key_1.
query secret active_master_key.
query secret sa_key.
query secret encryption_key.
query secret decryption_key.
query secret mac.
query secret mac1.
query secret mac2.
query secret sa.
query secret sequence_number.
query secret l4_hdr.
query secret l4_payload.
query secret ip_payload.
query secret P.
query secret pkt_in.
query secret pkt.
query secret pkt_out_rx.
query secret pkt_PspHdrStart_PktLen.
```

```
21 query secret pkt_CryptOffset_IcvStart.
22
23 (* Result: false *)
24 query secret spi.
25 query secret spi_msb.
26 query secret spi_31lsb.
27 query secret crypt_offset.
28 query secret ip_hdr.
29 query secret ip_payload_0_CryptOffset.
30 query secret psp_hdr.
31 query secret psp_hdr_no_iv.
32 query secret psp_hdr_offset.
33 query secret iv.
34 query secret gcm_iv.
35 query secret icv.
36 query secret C.
37 query secret A.
38 query secret pkt_out.
39 query secret pkt_in_rx.
40 query secret pkt_rx.
41 query secret pkt_0_PspHdrStart.
42 query secret pkt_0_CryptOffset.
43 query secret pkt_PspHdrStart_IcvStart.
44 query secret pkt_PspHdrStart_PktLen_enc.
```

Listing C.1: Secrecy queries.

C.2 Queries to test isolation between connections

The queries for the isolation tests are located within separate ProVerif programs, psp_leak_master_key.pv and psp_leak_sa_key.pv, but are exactly the same. In these programs, we intentionally leak some key parameters from one connection and see whether this affects the secrecy and authenticity of the payload from another connection. In the first program, we set up two connections using different master keys and then expose the master keys in one of them. In the second program, we set up two connections using the same master keys and then expose the SA key in one of them. The queries used to test for isolation are shown in Listing C.2. Lines 2 and 4 checks the secrecy of the payload in the compromised connection, while lines 3 and 5 do the same for the unaffected connection. Similarly, lines 8-12 checks the authenticity of

the payload in the compromised connection, while lines 14-18 do the same for the unaffected connection. The results show that the secrecy and authenticity of the payload is preserved in the connection that has not been compromised, proving that the isolation property is indeed satisfied by our model.

```
1  (* Secrecy tests *)
2  query secret P.
3  query secret P_alt.
4  query secret pkt_CryptOffset_IcvStart.
5  query secret pkt_CryptOffset_IcvStart_alt.
6
7  (* Authenticity tests *)
8  query spi: bitstring, mk0: mkey_t, mk1: mkey_t, sa: sa_t, k: skey_t,
       iv: bitstring, p: bitstring, c: bitstring, a: bitstring, icv:
     bitstring;
9    event(upper_layer_data_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
     ));
10   event(upper_layer_data_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
     )) ==> (event(psp_data_with_spi_tx(spi, sa, k, iv, p, c, a, icv))
      ==> (event(end_sa_0_128(mk0, mk1, sa)) ==> event(end_master_key(
     mk0, mk1))));
11   event(upper_layer_data_rx_1_128(spi, mk0, mk1, k, iv, p, c, a, icv
     ));
12   event(upper_layer_data_rx_1_128(spi, mk0, mk1, k, iv, p, c, a, icv
     )) ==> (event(psp_data_with_spi_tx(spi, sa, k, iv, p, c, a, icv))
      ==> (event(end_sa_1_128(mk0, mk1, sa)) ==> event(end_master_key(
     mk0, mk1)))).
13
14 query spi: bitstring, mk0: mkey_t, mk1: mkey_t, sa: sa_t, k: skey_t,
       iv: bitstring, p: bitstring, c: bitstring, a: bitstring, icv:
     bitstring;
15   event(upper_layer_data_rx_0_128_alt(spi, mk0, mk1, k, iv, p, c, a,
       icv));
16   event(upper_layer_data_rx_0_128_alt(spi, mk0, mk1, k, iv, p, c, a,
       icv)) ==> (event(psp_data_with_spi_tx_alt(spi, sa, k, iv, p, c,
     a, icv)) ==> (event(end_sa_0_128_alt(mk0, mk1, sa)) ==> event(
     end_master_key_alt(mk0, mk1))));
17   event(upper_layer_data_rx_1_128_alt(spi, mk0, mk1, k, iv, p, c, a,
       icv));
18   event(upper_layer_data_rx_1_128_alt(spi, mk0, mk1, k, iv, p, c, a,
       icv)) ==> (event(psp_data_with_spi_tx_alt(spi, sa, k, iv, p, c,
     a, icv)) ==> (event(end_sa_1_128_alt(mk0, mk1, sa)) ==> event(
```

```
    end_master_key_alt(mk0, mk1)))).
```

Listing C.2: Queries for test isolation between connections.

C.3 Queries and trace graphs for concurrency tests

We elaborate on the concurrency tests introduced in Section 6.8. To show that our PSP program is rich enough to model different scenarios involving multiple transactions and connections, we designed a series of reachability queries to simulate different scenarios involving concurrency. Executing these queries would then output trace graphs which confirm that these scenarios can be modelled by our program.

C.3.1 Consecutive transmissions within same connection

The query to model this scenario is shown in Listing C.3. This query induces ProVerif to produce a trace graph containing two of the same events with the same master keys, SA key, SPI, but different IVs. Upon execution, ProVerif produces a trace graph which looks like the simplified version in Figure C.1. The complete trace graph can be found in the psp_results folder. This proves that our PSP model supports consecutive transmissions within the same connection.

```
1 query spi: bitstring, mk0: mkey_t, mk1: mkey_t, k: skey_t, iv:
    bitstring, p: bitstring, c: bitstring, a: bitstring, icv:
    bitstring, iv': bitstring, p': bitstring, c': bitstring, a':
    bitstring, icv': bitstring;
2   event(upper_layer_data_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
    )) && event(upper_layer_data_rx_0_128(spi, mk0, mk1, k, iv', p',
    c', a', icv')) && (iv <> iv');
3   event(upper_layer_data_rx_1_128(spi, mk0, mk1, k, iv, p, c, a, icv
    )) && event(upper_layer_data_rx_1_128(spi, mk0, mk1, k, iv', p',
    c', a', icv')) && (iv <> iv').
```

Listing C.3: Queries to simulate consecutive transmissions within same connection.

C.3.2 Concurrent connections using same master keys but different SA keys

The query to model this scenario is shown in Listing C.4. This query induces ProVerif to produce a trace graph containing two of the same events with the same master keys

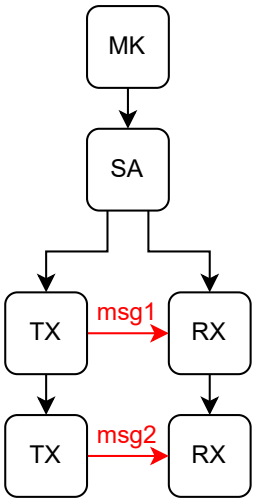

Figure C.1: Consecutive transmissions within same connection.

but different SA keys. Upon execution, ProVerif produces a trace graph which looks like the simplified version in Figure C.2. The complete trace graph can be found in the psp_results folder. This proves that our PSP model supports concurrent connections using same master keys and different SA keys.

```
1 query spi: bitstring, mk0: mkey_t, mk1: mkey_t, k: skey_t, iv:
    bitstring, p: bitstring, c: bitstring, a: bitstring, icv:
    bitstring, spi': bitstring, k': skey_t, iv': bitstring, p':
    bitstring, c': bitstring, a': bitstring, icv': bitstring;
2   event(upper_layer_data_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
    )) && event(upper_layer_data_rx_0_128(spi', mk0, mk1, k', iv', p
    ', c', a', icv')) && (k <> k');
3   event(upper_layer_data_rx_1_128(spi, mk0, mk1, k, iv, p, c, a, icv
    )) && event(upper_layer_data_rx_1_128(spi', mk0, mk1, k', iv', p
    ', c', a', icv')) && (k <> k').
```

Listing C.4: Queries to simulate concurrent connections using same master keys but different SA keys.

C.3.3 Concurrent connections using master keys 0 and 1 on same receiver

The query to model this scenario is shown in Listing C.5. This query induces ProVerif to produce a trace graph containing two of the same events with the same master keys but each with a different active master key. Upon execution, ProVerif produces a trace graph

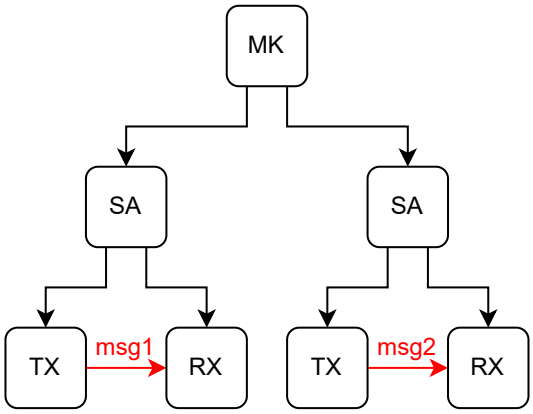

Figure C.2: Concurrent connections using same master keys but different SA keys.

which looks like the simplified version in Figure C.3. The complete trace graph can be found in the psp_results folder. This proves that our PSP model supports concurrent connections using master keys 0 and 1 on same receiver.

```
1 query spi: bitstring, mk0: mkey_t, mk1: mkey_t, k: skey_t, iv:
    bitstring, p: bitstring, c: bitstring, a: bitstring, icv:
    bitstring, spi': bitstring, k': skey_t, iv': bitstring, p':
    bitstring, c': bitstring, a': bitstring, icv': bitstring;
2   event(upper_layer_data_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
    )) && event(upper_layer_data_rx_1_128(spi', mk0, mk1, k', iv', p
    ', c', a', icv')) && (k <> k');
3   event(upper_layer_data_rx_1_128(spi, mk0, mk1, k, iv, p, c, a, icv
    )) && event(upper_layer_data_rx_0_128(spi', mk0, mk1, k', iv', p
    ', c', a', icv')) && (k <> k').
```

Listing C.5: Queries to simulate concurrent connections using master keys 0 and 1 on same receiver.

C.3.4 Concurrent connections using different master keys on different receivers

The query to model this scenario is shown in Listing C.6. This query induces ProVerif to produce a trace graph containing two of the same events with different master keys. Upon execution, ProVerif produces a trace graph which looks like the simplified version in Figure C.4. The complete trace graph can be found in the psp_results folder. This proves that our PSP model supports concurrent connections using different master keys on different receivers.

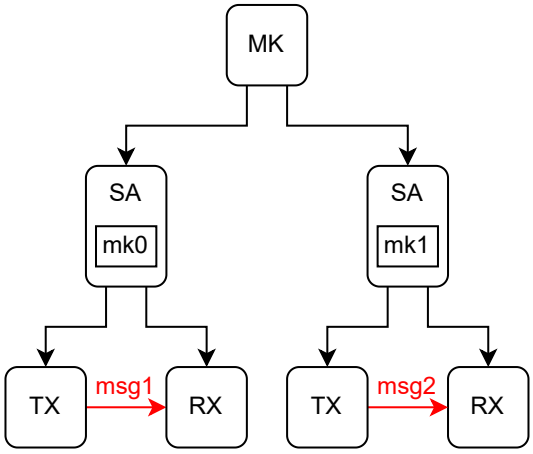

Figure C.3: Concurrent connections using master keys 0 and 1 on same receiver.

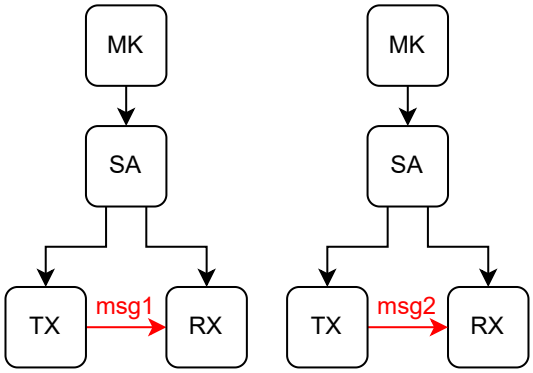

Figure C.4: Concurrent connections using different master keys on different receivers.

```
1  query spi: bitstring, mk0: mkey_t, mk1: mkey_t, k: skey_t, iv:
       bitstring, p: bitstring, c: bitstring, a: bitstring, icv:
       bitstring, spi': bitstring, mk0': mkey_t, mk1': mkey_t, k':
       skey_t, iv': bitstring, p': bitstring, c': bitstring, a':
       bitstring, icv': bitstring;
2    event(upper_layer_data_rx_0_128(spi, mk0, mk1, k, iv, p, c, a, icv
       )) && event(upper_layer_data_rx_0_128(spi', mk0', mk1', k', iv',
       p', c', a', icv')) && (mk0 <> mk0') && (mk1 <> mk1');
3    event(upper_layer_data_rx_1_128(spi, mk0, mk1, k, iv, p, c, a, icv
       )) && event(upper_layer_data_rx_1_128(spi', mk0', mk1', k', iv',
       p', c', a', icv')) && (mk0 <> mk0') && (mk1 <> mk1').
```

Listing C.6: Queries to simulate concurrent connections using different master keys on different receivers.

Appendix D

Complete list of programs and results

Table D.1 shows the list of files developed in this project, a description of their functions, and the commands used to execute the programs using ProVerif. The listed commands are assumed to be executed in the directory containing the ProVerif binary named "proverif". All library and program files are assumed to be residing within this directory. All the program files are using messages with single blocks, unless stated otherwise.

File name	**Description**	**Command to execute program**
utils.pvl	Library file that defines constants, functions, and equations common to all programs.	
cmac.pvl	Library file that defines the CMAC model and its operations.	
cmac_no_k1.pvl	Library file that defines a flawed CMAC model without subkey derivation.	
cmac_sender_verifier.pv	Program file that models the transmission of a message-tag pair from sender to verifier using CMAC.	proverif -lib utils -lib cmac cmac_sender_verifier.pv

Table D.1 continued from previous page

cmac_sender_verifier2.pv	Same as above but with different number of message blocks to demonstrate the length-extension attack.	proverif -lib utils -lib cmac_no_k1 cmac_sender_verifier2.pv
cmac_euf-cma.pv	Program file that models the EUF-CMA game.	proverif -lib utils -lib cmac cmac_euf-cma.pv
gcm.pvl	Library file that defines the GCM model and its operations.	
gcm_no_final_xor.pvl	Library file that defines a flawed GCM model without the final XOR.	
gcm_no_length.pvl	Library file that defines a flawed GCM model without the length function.	
gcm_sender_receiver.pv	Program file that models the transmission of a message from sender to receiver using GCM. Also used to demonstrate the message forgery attack.	proverif -lib utils -lib gcm gcm_sender_receiver.pv, proverif -lib utils -lib gcm_no_final_xor gcm_sender_receiver.pv
gcm_sender_receiver2.pv	Same as above but with different number of message blocks to demonstrate the length-extension attack.	proverif -lib utils -lib gcm_no_length gcm_sender_receiver2.pv
gcm_ind-cca2.pv	Program file that models the IND-CCA2 game.	proverif -lib utils -lib gcm gcm_ind-cca2.pv
psp.pv	Program file that models the transmission of a single-block packet from transmitter to receiver using PSP.	proverif -lib utils -lib cmac -lib gcm psp.pv

Table D.1 continued from previous page

psp2.pv	Program file that models the transmission of a double-block packet from transmitter to receiver using PSP.	proverif -lib utils -lib cmac -lib gcm psp2.pv
psp_leak_master_key.pv	Program file that intentionally leaks the master keys in one of the processes using PSP.	proverif -lib utils -lib cmac -lib gcm psp_leak_master_key.pv
psp_leak_sa_key.pv	Program file that intentionally leaks the SA key in one of the processes using PSP.	proverif -lib utils -lib cmac -lib gcm psp_leak_sa_key.pv

Table D.1: ProVerif files created in this project.

Table D.2 lists the folders containing the query results from executing the ProVerif program files. The results have been formatted in HTML and trace graphs have also been created for the relevant queries. The recommended way to view the results is to run the index.html file located within each folder. This will then display the results on a browser using a layout that is easy to navigate.

Folder name	**Description**
cmac_sender_verifier_results	Results from executing cmac_sender_verifier.pv.
cmac_euf-cma_results	Results from executing cmac_euf-cma.pv.
cmac_no_k1_length_extension_attack	Results from executing cmac_sender_verifier2.pv.
gcm_sender_receiver_results	Results from executing gcm_sender_receiver.pv.
gcm_ind-cca2_results	Results from executing gcm_ind-cca2.pv.
gcm_no_final_xor_message_forgery	Results from executing gcm_sender_receiver.pv with gcm_no_final_xor.pvl.

Table D.2 continued from previous page

gcm_no_length_length_extension_attack	Results from executing gcm_sender_receiver2.pv.
psp_results	Results from executing psp.pv.
psp2_results	Results from executing psp2.pv.
psp_leak_master_key_results	Results from executing psp_leak_master_key.pv.
psp_leak_sa_key_results	Results from executing psp_leak_sa_key.pv.

Table D.2: Folders containing results from executing the ProVerif programs.