Cryptographic Security of SSH Encryption Schemes

Torben Brandt Hansen

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Department of Information Security
Royal Holloway, University of London

2020

Declaration

These doctoral studies were conducted under the supervision of Prof. Kenneth G. Paterson.

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Information Security as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Torben Brandt Hansen
June, 2020

Acknowledgements

A grateful thanks to Kenny Paterson for his great supervision and supporting my non-academic endeavours during my Ph.D. Even more for his mentorship and for pushing me to develop and improve my academic skills; it is impossible to leave Kenny's office without learning something new and exciting. And, not least, thank you for your dedicated attempts to cryptographically hide***s*** my silly grammar mistakes.

Also a special thanks to my advisor Martin Albrecht for always providing support and great collaborations.

Thank you to Royal Holloway and EPSRC for providing the funding, allowing me to pursue a Ph.D. and for given me many opportunities to explore the real world.

Most importantly, I am in debt to Aeriel for supporting me through my Ph.D. and encouraging me to pursue my dreams. *Mahal Kita*

Abstract

SSH is a Swiss Army Knife protocol for creating secure communication links between machines and an indispensable tool for IT professionals. However, its underlying symmetric encryption scheme constructions have not seen the same rigorous analysis as constructions in other popular secure communication protocols. This thesis aims to bridge this gap, providing SSH deployment statistics, new attacks against a number of SSH encryption schemes, a thorough security analysis of several SSH encryption schemes, and development of new SSH encryption schemes that provide better security properties than existing schemes.

Firstly, we report on several scans performed targeting publicly accessible SSH servers on the Internet. From these scans, we compile longitudinal SSH statistics evaluating the evolution of preferred SSH encryption scheme, SSH software and SSH version trends.

Secondly, we describe several new attacks on SSH encryption schemes in OpenSSH that utilise the CBC encryption mode of operation. These attacks are a result of both inherent weaknesses in CBC-mode and bugs in the OpenSSH implementation.

Thirdly, we use the ciphertext fragmentation framework to analyse the concrete cryptographic security of a number of SSH encryption schemes as implemented in OpenSSH.

Finally, we develop a practical version of the symmetric encryption scheme *InterMAC*, implement it and evaluate its security and performance. The implementation is then used to construct InterMAC-based SSH encryption schemes. We evaluate the performance of these new schemes against existing SSH encryption schemes in OpenSSH.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives an overview of the thesis. We provide the motivation for our research and describe the contributions of this thesis. In this chapter, we also present the overall structure of the thesis.

1.1 Motivation

Authenticated Encryption (AE) security has emerged as the standard security notion that a symmetric encryption scheme should satisfy to be considered for practical use. AE security is equivalent to achieving IND-CPA and INT-CTXT security, meaning confidentiality against a passive attacker and integrity against an active attacker armed with a decryption capability. However, AE security is not sufficient for every application scenario. A case in point is SSH,[1] specifically the Binary Packet Protocol, the component of the SSH protocol suite specifying data transfer. SSH continues to be an indispensable tool for system administrators. Originally designed as a secure replacement for unencrypted protocols such as *Telnet* and *rsh*, it has since established itself as the primary protocol for remote login to UNIX environments and has been extended to cover bulk file transfers and other applications. Its importance is underlined by Microsoft making OpenSSH, a popular SSH implementation, available on its Microsoft Windows 10 operating system.[2]

In 2002 Bellare et al. [20] proved that the variant of "Encrypt & Mac" used in SSH provides (stateful) AE security under reasonable assumptions on the protocol's building blocks. However, in 2009 Albrecht et al. [5] presented a plaintext-recovery attack against SSH encryption schemes using CBC-mode with random IVs in SSH,

[1] The current version of the protocol is version two and is denoted SSHv2, but we will write SSH as a shorthand.

[2] `https://www.zdnet.com/article/openssh-arrives-in-windows-10-spring-update` (accessed 12/02/2020).

a case covered by the proof. The attack exploits the fact that ciphertexts can be delivered as a sequence of fragments, with the attacker being able to observe the behaviour of the receiver as each fragment is delivered. The attack also exploits the fact that SSH tries to hide information about packet lengths by encrypting the relevant length field. Traditional security notions, including those used by Bellare et al. [20], do not cater for this kind of *ciphertext fragmentation attack*. Fragmentation of ciphertexts has also been used to mount attacks against IPSec [55].

The attack in [5] motivated significant follow-up work in two distinct yet closely coupled directions — theoretical modelling and the widespread deployment of improved encryption schemes in SSH.

In the first direction, Paterson and Watson [114] analysed SSH's use of CTR-mode, showing that it achieves security in a prototypical model supporting ciphertext fragmentation. The paper [114] inspired the more general and mature treatment of *symmetric encryption supporting fragmented decryption* by Boldyreva *et al.* [36]. That paper introduced general notions formalising confidentiality against fragmentation attacks (IND-sfCFA). It also showed that more advanced security notions considered desirable by the designers of SSH, namely boundary hiding security (BH-CPA and BH-sfCFA) and resistance to certain types of denial-of-service attack (DOS-sfCFA), could be achieved at the same time as confidentiality at low cost using only standard tools. In this thesis, we expand the set of notions to an integrity notion (INT-sfCTF) that can also be realised in combination with the rest of the ciphertext fragmentation notions, in the same way.

In the second direction, deployment of improved SSH encryption schemes, we have seen a proliferation of alternative choices of encryption schemes being introduced to SSH. First, the OpenSSH implementation of CBC-mode was quickly patched to prevent the specific attack in [5]. Then many implementations (including OpenSSH and the lightweight SSH implementation Dropbear, popular on embedded devices) moved to make CTR-mode the default choice, a selection well-supported by the analysis in [114]. More recently, SSH-AES-GCM, generic Encrypt-then-MAC constructions and SSH-ChaCha20-Poly1305 have been added to OpenSSH. However, except for CTR-mode, these new modes have not yet been subjected to any serious scrutiny by the research community *in the SSH context*, where, recall, additional attack capabilities beyond those usually assumed in the symmetric encryption setting must be taken into account.

Given the serious nature of the attacks in [5], and the sensitive, high-value nature of at least some traffic protected by SSH, we assert that it is an important problem to study whether the newly introduced SSH encryption schemes do lead to a secure channel in SSH. This is also a particularly timely problem to address, in view of SSH-ChaCha20-Poly1305, one of the new schemes, having been promoted to default algorithms in OpenSSH 6.9 in mid-2015, and is now a particular popular choice.

In this thesis, we address this problem and bring the two directions described above — theoretical modelling and deployment of improved choices of encryption schemes— together. We present a systematic analysis of popular symmetric encryption schemes for SSH and deployment statistics for both SSH encryptions schemes and SSH software products. In addition, we also show that our systematic analysis is timely by presenting several new attacks against the CBC-mode encryption schemes in OpenSSH.

However, as our results show, it is notable that none of the symmetric encryption schemes currently supported in SSH (nor in the leading OpenSSH implementation) achieve the four strongest properties in combination (IND-sfCFA, INT-sfCTF, BH-sfCFA, DOS-sfCFA). For example, the now-default SSH encryption scheme in OpenSSH SSH-ChaCha20-Poly1305, is based on ChaCha20-Poly1305, but uses two separate keys, one for encrypting the length field, and another for encrypting actual data. The length field encryption highlights the OpenSSH developers' desire to achieve some form of boundary hiding, a desire explicitly confirmed by one of the main OpenSSH developers, Damien Miller [110]. The SSH-ChaCha20-Poly1305 scheme still suffers from weaknesses that lead to easy attacks: an attacker can manipulate the length field, enabling a DOS-sfCFA attack; meanwhile BH-sfCFA attacks are possible by "bit-flipping" elsewhere in the packet and observing how many bytes of input are needed to trigger MAC errors. The scheme does at least achieve BH-CPA security.

The above discussion begs the question: can one do better, achieving all four of the strong security notions formalised by Boldyreva et al. [36], and at what cost? In fact, [36] already defined a scheme, InterMAC, that meets the four security notions: IND-sfCFA, INT-sfCTF, BH-sfCFA, and DOS-sfCFA. Their scheme breaks a message into equal-sized chunks and applies an Encrypt-then-MAC construction to them separately, incorporating certain encoding information in the MAC computation to indicate the final chunk of a message and to ensure that chunks cannot be reordered or deleted. The size of each chunk in bytes, called the chunk length, is a parameter of the scheme which we denote by N. It determines the amount of DoS security the InterMAC scheme offers: it is guaranteed that decryption must output either some plaintext or an error message for every $N + \delta$ bytes of ciphertext received, where δ is some small overhead (related to the scheme's ciphertext expansion).

Reflecting on the discussion above, it is evident that there is a need, beyond analysing existing SSH encryption schemes, to implement and deploy schemes that satisfy the advanced security notions desired (but not currently met) by the designers of the SSH protocol. To rectify this situation, this thesis develops a practical version of InterMAC and uses this version to implement new InterMAC-based SSH encryption schemes for OpenSSH.

1.2 Vulnerability Disclosure

We disclosed our new attacks to the OpenSSH team in two different disclosures. We first notified the OpenSSH team of our second new attack on CBC-mode on 5/5/2016. Our first and third new attacks on CBC-mode were notified to the OpenSSH team on 15/12/2016. The first attack was applicable to OpenSSH version 5.2 - 7.4 and was addressed in OpenSSH version 7.5, released 20/3/2017.[3] The second attack was applicable to OpenSSH versions 5.2 - 7.2 and addressed in OpenSSH version 7.3, released 1/8/2016.[4] The third attack has been applicable since OpenSSH version 7.3. We also notified the OpenSSH team of a flaw in MAC processing in the SSH-Generic-EtM scheme on 22/5/2016. This issue was also addressed in OpenSSH version 7.3.

1.3 Organisation of Thesis

The remainder of this thesis is organised as follows.

Chapter 2. In this chapter, we give an overview of modern cryptography, introducing key concepts and definitions used throughout. We also give an introduction to the parts of the SSH protocol that will become relevant in later chapters.

Chapter 3. This chapter introduces the ciphertext fragmentation model, covering

[3] See `http://www.openssh.com/txt/release-7.5`.

[4] See `http://www.openssh.com/txt/release-7.3`.

notation and definitions. We analyse the model and surface a number of critical points. Furthermore, we fix a flaw in the original confidentiality definition from [36]. At the end of the chapter, we compare the ciphertext fragmentation model to similar security models in the literature.

Chapter 4. In this chapter, we present statistics on the SSH ecosystem on the Internet collected through three scans performed in 2015, 2016 and 2019. In our presentation, we focus on the different SSH implementations and versions, as well as the SSH encryption schemes preferred by SSH servers. In addition, we analyse several SSH software identifiers returned by SSH servers, attempting to identify the underlying software.

Chapter 5. This chapter describes three new attacks against SSH encryption schemes in OpenSSH that use CBC-mode. We describe in detail how each attack is performed, how each attack could potentially be mitigated, and the practical impact of the attacks. Furthermore, we explain how an old attack by Albrecht, Paterson and Watson [5] can be applied to Dropbear.

Chapter 6. In this chapter, we provide a formal treatment of several SSH encryption schemes, as implemented in OpenSSH, in the ciphertext fragmentation model. We show that many schemes meet confidentiality and integrity notions, but most schemes fail to meet the more advanced notions of boundary hiding and denial-of-service resistance.

Chapter 7. This chapter introduces an implementation of the InterMAC scheme. The scheme is first modified with an aim towards making it usable in practice. We then present libInterMAC, a library that implements the InterMAC scheme. We describe several implementation techniques used to mitigate potential side-channel attack vectors. We use libInterMAC to implement InterMAC-based SSH encryption schemes in OpenSSH. Finally, we provide a thorough performance analysis of both libInterMAC and the InterMAC-based SSH encryption schemes, and compare them against existing SSH schemes in OpenSSH.

1.4 Associated Publications

Chapters 3, 4, 5 and 6 are based on joint work with Martin R. Albrecht, Jean Paul Degabriele and Kenneth G. Paterson [A], as well as unpublished work developed jointly with the same authors. Chapter 7 is based on joint work with Martin R. Albrecht and Kenneth G. Paterson [B]. All authors contributed equally to the above publications.

[A] Martin R. Albrecht, Jean Paul Degabriele, Torben Brandt Hansen, and Kenneth G. Paterson. A surfeit of SSH cipher suites. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 1480-1491. ACM Press, October 2016.

[B] Martin R. Albrecht, Torben Brandt Hansen, and Kenneth G. Paterson. libInterMAC: Beyond confidentiality and integrity in practice. *IACR Transactions on Symmetric Cryptology*, 2019(1):46-83, 2019.

Chapter 2

Background

This chapter covers background material, highlights key assumptions and fixes notation.

2.1 Notation

We let $\mathsf{Func}(\mathcal{X}, \mathcal{Y})$ denote the set of functions from set $\mathcal{X}$ to set $\mathcal{Y}$. For algorithms $\mathsf{A}_1, \mathsf{A}_2, \ldots$, we let $\mathcal{A}^{\mathsf{A}_1, \mathsf{A}_2, \cdots}$ denote the output after executing $\mathcal{A}$ with oracle access to $\mathsf{A}_1, \mathsf{A}_2, \ldots$ and with fresh coins. We use $\{0,1\}^*/\mathsf{B}^*$ to denote the set of all bit/byte strings of finite length and $\{0,1\}^n/\mathsf{B}^n$ to denote the set of all bit/byte strings of length n. If S is a set, then S^+ denotes the set of all combinations of concatenations of elements from S and $s \leftarrow\!\!\$\, S$ means sampling an element s uniformly at random from S. If i is an unsigned integer, $\langle i \rangle_\ell$ denotes the unsigned ℓ-bit representation of i (truncating the most significant bits if the bit length of i is larger than ℓ). Accordingly, $\langle \cdot \rangle^{-1}$ represents the inverse mapping which maps strings of any length to $\mathbb{N}$. ϵ denotes the empty string.

We view a string v in two ways: either as a string consisting of *bits* or as a string consisting of *bytes*. In the former case, we call v a *bit-string*, while in the latter case, we call v a *byte-string*. This defines two type of strings, and we refer to this as v's *string type*. A string can only be a byte-string if its length, counted in bits, is a multiple of 8. A byte-string is also a bit-string, while a bit-string is not necessarily a byte-string. The specific byte-notation, indicated by the subscript B, presented below can only be used for byte-strings. All bit-wise operations defined below can also be performed on byte-strings by viewing the byte-string as a bit-string. For two strings v and w, with the same string type, we use the following notation:

- $v \parallel w$ denotes the concatenation of strings v and w.

- $|v|$ denotes the size of string v counted in bits.
- $|v|_{\mathsf{B}}$ denotes the size of string v counted in bytes.
- $v \mid w$ denotes the bit-wise logical OR operation between v and w (of the same length).
- $v \oplus w$ denotes the bit-wise logical XOR operation between v and w (of the same length).
- $v \mathbin{\&} w$ denotes the bit-wise logical AND operation between v and w (of the same length).
- $\sim v$ denotes the bit-wise logical negation of v.
- $v \gg l$ denotes the right-shift of v by l bits.
- $v \preceq w$ denotes the prefix predicate and returns true if and only if there exists $u \in \{0,1\}^*$ such that $w = v \parallel u$, where u must have the same string type as v and w.
- $v \,\%\, w$ denotes the (unique) string z such that $v = p \parallel z$, where $p \in \{0,1\}^*$ is the greatest common prefix of v and w (i.e. p is the longest string such that $p \preceq v$ and $p \preceq w$), where p and z must have the same string type as v and w.
- $v[i]$ denotes the ith bit of v. Indexing starts at 0 and no index below 0 is allowed.
- $v[i]_{\mathsf{B}}$ denotes the ith byte of v. Indexing starts at 0 and no index below 0 is allowed.
- $v[i:j]$ denotes the substring from bit i to bit j (inclusive) of v.
- $v[i:j]_{\mathsf{B}}$ denotes the substring from byte i to byte j (inclusive) of v.

For a list $\mathcal{L}$, we use the following notation:

- $\mathcal{L} = []$ denotes the initialisation of $\mathcal{L}$ to the empty list.
- $\mathcal{L}.\mathsf{append}(L)$ denotes appending the element L to the list $\mathcal{L}$.
- $L \in \mathcal{L}$ denotes the list membership test and returns true if L is an element in the list $\mathcal{L}$ and false otherwise.
- $\mathcal{L}[i]$ denotes the ith element of $\mathcal{L}$. Indexing start at 0 and no index below 0 is allowed.
- $\mathcal{L}[p \dots q]$ denotes the sublist $[\mathcal{L}[p], \mathcal{L}[p+1], \dots, \mathcal{L}[q]]$. If $q < p$ the result is the empty list $[]$.

- $|\mathcal{L}|$ denotes the number of elements in the list $\mathcal{L}$. The number of elements in the empty list is 0.
- If the elements of $\mathcal{L}$ are (all) strings, where all strings must be of the same string type, then $||(\mathcal{L})$ denotes the concatenation $\mathcal{L}[0] \parallel \mathcal{L}[1] \parallel \cdots \parallel \mathcal{L}[|\mathcal{L}| - 1]$. For the empty list, we use the convention $||([]) = \epsilon$.
- If S is a finite set, then $[S]^n$ and $[S]^*$ denotes the set of all lists of length n with elements from S and the set of all (finite) lists with elements from S, respectively.

We use $\Pr[x \leftarrow P \,:\, \phi(x)]$ to denote the probability that after having executed process P (which might be running a probabilistic algorithm or drawing elements from certain distributions (or both)) returning output x, the predicate $\phi(x)$ is true. ϕ is allowed to execute probabilistic algorithms. The output x will often be suppressed in the presentation. Security definitions use notation from the code-based game-playing framework (cf. [24]) with some minor changes in presentation, e.g. we will never use explicit finalise procedures.

Given a security game G (cf. Section 2.3.2) and an adversary $\mathcal{A}$, we use $\mathsf{G}(\mathcal{A})$ to mean that $\mathcal{A}$ is playing the game G.

2.2 Bits vs Bytes

In this thesis, we take a byte-oriented approach when describing algorithms. That is, algorithms use byte-wise operations. However, because we want to be compatible with standard works, we keep standard bit-oriented security definitions.

Our main reason for choosing byte-oriented notation is the strong practical view we take. Odd bit lengths are a significant headache for implementers[1] and seldom used in network protocols, SSH being a prime example of a byte-oriented protocol. As a result, describing data strings, etc., in bits (not 8-bit multiples) imposes extra work and creates confusion, but yields no practical advantages.

There are certainly systems for which operating on bits can be beneficial. For example, real-time wireless sensor networks for, e.g., fire monitoring [136] or border surveillance [44]. Such systems usually demand not only high data integrity assurance

[1] As an example, in the native C-language types are all byte oriented

but also high timeliness [129]. Due to deployment environments, this has to be met with strict restrictions on bandwidth and energy consumption, making it necessary to consider non-byte formats.

All algorithm descriptions in this thesis can be converted to a bit representation, with relative ease.

2.3 Modern Cryptography

This section introduces some of the theory of modern cryptography. The purpose of this section is to fix notation and definitions used throughout. The treatment is somewhat cursory, and we will not dive very deep into the fascinating theory of modern cryptography. For a more thorough treatment of modern cryptography the reader can consult e.g. [93] (appropriately named *Introduction to Modern Cryptography*).

2.3.1 Concrete Security

To quantify security of cryptographic schemes, we utilise the *concrete security* approach developed by Bellare [14] and Rogaway [121] as part of their push for practice-oriented provable-security. In this approach the security of a cryptographic scheme is *concretely* quantified in terms of its components, and the resources consumed by an adversary. The goal is to give results that are meaningful in practice. To this extent, a definition in the concrete security model takes roughly the following form:

A scheme is (η, R)*-secure, if for any adversary, with*
resources at most R*, its success probability is bounded by* η *(from above).*

In this thesis, we use game-based definitions of security. That is, when we state that a scheme is secure, we mean that the scheme meets a security property encapsulated through a game played by an adversary. The adversary can "win" the game by satisfying a particular condition. The success probability is derived through this game, by computing the probability of an adversary succeeding in satisfying the condition. We will often refer to the success probability as the adversary's *advantage*.

The amount of resources consumed by an adversary is also derived through the game. Specific resources considered might change depending on the context, but we will always consider a subset of the following resources: number of queries the adversary makes, the total length (in bits or bytes) of its queries (and responses), and the computation time of the adversary.

Another approach often used when analysing cryptographic schemes is the asymptotic approach. In this approach, polynomial-time adversaries are considered, and security is quantified over families of schemes parameterised by a *security parameter*. Security of a scheme can be scaled by adjusting the security parameter.

Computational complexity is measured in an underlying model that specifies how to measure the computational time of an adversary. Such a model will not be defined here. We assume that adversaries and games terminate in finite time. This, in turn, implies that both adversaries and games must draw a finite number of random elements and hence, the underlying sample space is finite.

For more details on the concrete security approach the reader is encouraged to consult [18, 19, 23].

2.3.2 Proofs and Security Notions

As mentioned earlier, security properties will be captured through games played by an adversary. We call such a game a *security game*. An adversary wins a security game if the win condition is satisfied, the adversary's advantage is "high", and there is a "reasonable" upper bound on the amount of resources consumed. What is considered a reasonable amount of resources is purposely left undefined because the tolerable power of an adversary depends on the practical context. For the same reason, what is considered a high advantage is also left undefined.

To prove a security property, we prove that an adversary can only win the associated security game (i.e meet the win condition) with "low" advantage when using a reasonable amount of resources. To show this, we use proofs by reduction. To illustrate this technique, consider the Discrete Logarithm problem (DL) and the Decisional Diffie Hellman problem (DDH) [39]. Informally, in the former problem an adversary should be unable to compute the discrete logarithm while in the latter problem an adversary should be unable to distinguish between tuples (g^x, g^y, g^z) and (g^x, g^y, g^{xy}) (using standard notation). These problems can be viewed as security games. Given an adversary $\mathcal{A}_{\mathsf{DL}}$ that solves the DL problem, we can construct a "reduction" $\mathcal{A}_{\mathsf{DDH}}$ that solves the DDH problem. This shows that if we can solve DL, then we can also solve DDH. In other words, the DL problem is at least as hard as the DDH problem (i.e. there might be other ways to solve DDH). Note that the "quality" of the reduction matters. The quality of the reduction is measured by the amount of resources consumed by the reduction. Going back to our DL/DDH example, if we constructed $\mathcal{A}_{\mathsf{DDH}}$ such that it consumes exponentially more resources than $\mathcal{A}_{\mathsf{DL}}$, then the reduction does not necessarily say anything meaningful about $\mathcal{A}_{\mathsf{DDH}}$'s ability to solve the DDH problem in practice, in turn, nor does it say anything meaningful about the relationship between DDH and DL. This can lead to false conclusions about practical security levels of cryptographic constructions [42, 108, 43, 95].

A particular type (e.g. a probabilistic encryption scheme, cf. Section 2.3.4) of cryptographic construction together with an advantage definition and a security game defines a *security notion*. The security notion encapsulates the security property, how it is quantified, the information available to an adversary and what type of cryptographic construction the notion is valid for. Popular security notions are $\mathsf{IND\text{-}CPA}$ and $\mathsf{IND\text{-}CCA}$. In Chapter 3, we will define security notions that are less well known but capture stronger security properties. We will sometimes use the formulation that a cryptographic construction *meets* a security notion. This means that any adversary, using a "reasonable" amount of resources, cannot win the associated security game, but suppressing the explicit mention of the available resources and advantage.

All security notions are formulated in a single-user setting. This ignores the fact that many of today's systems are distributed and handle multiple users. A multi-user setting could, for example, be formulated using ideas from [16]. But we will not explore this further in this thesis.

Recall, that all security definitions below are all bit-oriented, while the algorithm descriptions, which will appear in later chapters, are byte-oriented.

2.3.3 Pseudorandom Functions

Consider a function $\mathsf{F}\colon \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$, with domain $\mathcal{X}$ and codomain $\mathcal{Y}$, where the first input is called the key and denoted by k. We only consider functions for which $\mathcal{K}$, $\mathcal{X}$ and $\mathcal{Y}$ are finite sets. For a specific key $\mathsf{k} \in \mathcal{K}$, we write $\mathsf{F}_{\mathsf{k}}(x) = \mathsf{F}(\mathsf{k}, x)$ for any $x \in \mathcal{X}$ and we say that F is a keyed function. If F is a function that is close to being a random function, we call it a pseudorandom function (PRF).

Definition 1 (PRF)**.**
Let $\mathsf{F}\colon \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ *be a keyed function. For an adversary* $\mathcal{A}$*, we define its* PRF *advantage (with respect to the keyed function* F*) as:*

$$\mathsf{Adv}_{\mathsf{F}}^{\mathsf{PRF}}(\mathcal{A}) = \Pr\left[\mathsf{k} \leftarrow_{\$} \mathcal{K} \,:\, \mathcal{A}^{\mathsf{F_k}(\cdot)} = 1\right] - \Pr\left[\mathsf{f} \leftarrow_{\$} \mathsf{Func}(\mathcal{X}, \mathcal{Y}) \,:\, \mathcal{A}^{\mathsf{f}(\cdot)} = 1\right].$$

The keyed function F *is* (η, R)-PRF *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* PRF *advantage is bounded by* η*.*

In this game, the adversary is given oracle access to either $\mathsf{F_k}$ or f, where k is chosen uniformly at random from $\mathcal{K}$ and f is chosen uniformly at random from $\mathsf{Func}(\mathcal{X}, \mathcal{Y})$. That is, a keyed function F is a PRF if the adversary is unable to distinguish the output from $\mathsf{F_k}$ and a random function f, even knowing input-output pairs.

PRFs were introduced by Goldreich, Goldwasser and Micali [75, 76] and are often used as building blocks in other cryptographic constructions. For example, symmetric encryption schemes (see Section 2.3.4) can be constructed from PRFs; PRFs are also frequently used to model the security of block ciphers [19], although we often want a stronger primitive, known as a PRP (where the adversary can query both F and its inverse F^{-1}), for this task (see Section 2.3.6).

2.3.4 Probabilistic Symmetric Encryption

In this section, we define the syntax of symmetric encryption schemes and define a set of security notions that capture security properties such schemes could aim to meet. Throughout a symmetric encryption scheme will consist of three algorithms: A key generation algorithm, an encryption algorithm and a decryption algorithm. For now, we will focus our attention on *probabilistic* symmetric encryption schemes. These are schemes in which the encryption algorithm is viewed as a randomised algorithm, choosing uniformly and independent random coins internally on each invocation. In Section 2.3.5, we will define symmetric encryption schemes which utilise either an initialisation vector or a nonce and instead view the encryption algorithm as being deterministic. We rely on this latter formulation a lot in this thesis.

Throughout this thesis, we will often abbreviate “probabilistic symmetric encryption scheme” with “symmetric encryption scheme” if the type of symmetric encryption scheme is clear from the context.

Syntax

Below we define the syntax of a symmetric encryption scheme. Note that we opted not to give a syntax that allows for multiple decryption errors. Allowing multiple decryption errors can have an effect on security [37].

Definition 2 (Probabilistic symmetric encryption scheme)**.**
A (probabilistic) symmetric encryption scheme $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with key space* $\mathcal{K} \subseteq \{0,1\}^*$*, plaintext space* $\mathcal{M} \subseteq \{0,1\}^*$*, ciphertext space* $\mathcal{C} \subseteq \{0,1\}^*$ *and error symbol* $\perp$ *(*$\perp \notin \mathcal{M} \cup \mathcal{C}$*) is specified by three algorithms:*

- *A randomised key generation algorithm* Gen *that outputs a key* $\mathsf{k} \in \mathcal{K}$*. We write* $\mathsf{k} \leftarrow \mathsf{Gen}$*.*
- *A randomised encryption algorithm* Enc *that takes as input a key* $\mathsf{k} \in \mathcal{K}$ *and a plaintext* $m \in \mathcal{M}$*, and outputs a ciphertext* $c \in \mathcal{C} \cup \{\perp\}$*. We write* $c \leftarrow \mathsf{Enc}_{\mathsf{k}}(m)$*.*
- *A deterministic decryption algorithm* Dec *that takes as input a key* $\mathsf{k} \in \mathcal{K}$ *and a ciphertext* $c \in \mathcal{C}$*, and outputs a plaintext* $m \in \mathcal{M} \cup \{\perp\}$*. We write* $m \leftarrow \mathsf{Dec}_{\mathsf{k}}(c)$*.*

We will sometimes omit the explicit mention of the plaintext space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ if they are not relevant. Both the encryption algorithm and the decryption algorithm are required to be stateless, and the encryption algorithm can also output errors. The latter is often omitted, but we feel that it best simulates practice, since encryption algorithms might fail for some reason (for example, if the encryption algorithm is fed a message not from $\mathcal{M}$).

Furthermore, we require that $\mathcal{M}$ contains at last two strings and that if $\mathcal{M}$ contains a string of length l, it contains all strings of length l. Finally, all symmetric encryption schemes are assumed to be length-regular. That is, if $c \leftarrow \mathsf{Enc}_{\mathsf{k}}(m)$ and $c \neq \perp$, then the length of c only depends on the length of m. This implies that all plaintext strings of length l map to ciphertexts of equal length.

It will become useful later to define a bit more notation such that we can more easily work on lists. For a list $\mathcal{L}_m = [m_1, m_2, \ldots, m_l] \in [\mathcal{M}]^l$, we write $\mathcal{L}_c \leftarrow \mathsf{Enc}_{\mathsf{k}}(\mathcal{L}_m)$ where $\mathcal{L}_c = [c_1, c_2, \ldots, c_l]$ and $c_1 \leftarrow \mathsf{Enc}_{\mathsf{k}}(m_1)$, $c_2 \leftarrow \mathsf{Enc}_{\mathsf{k}}(m_2), \ldots,$ $c_l \leftarrow \mathsf{Enc}_{\mathsf{k}}(m_l)$. For a list $\mathcal{L}_c = [c_1, c_2, \ldots, c_l] \in [\{0,1\}^*]^l$, we likewise write $\mathcal{L}_m \leftarrow \mathsf{Dec}_{\mathsf{k}}(\mathcal{L}_c)$ where $\mathcal{L}_m = [m_1, m_2, \ldots, m_l]$ and $m_1 \leftarrow \mathsf{Dec}_{\mathsf{k}}(c_1)$, $m_2 \leftarrow \mathsf{Dec}_{\mathsf{k}}(c_2), \ldots,$ $m_l \leftarrow \mathsf{Dec}_{\mathsf{k}}(c_l)$.

An important note about Definition 2 is the underlying assumption of *atomic* delivery of ciphertexts. This means that the decryption algorithm must operate on the entire ciphertext and cannot operate on parts of a ciphertext. In Chapter 3, we will discuss atomic delivery in detail.

Correctness

We require a symmetric encryption scheme to be perfectly correct: for all keys $\mathsf{k} \in \mathcal{K}$, that have a non-zero probability of being output by Gen and all $m \in \mathcal{M}$ the following is true with probability one:

$$\text{if } c \leftarrow \mathsf{Enc}_{\mathsf{k}}(m) \text{ and } c \neq \bot, \text{ then } m \leftarrow \mathsf{Dec}_{\mathsf{k}}(c).$$

We implicitly assume that the correctness property is satisfied by all symmetric encryption schemes used in this thesis.

Confidentiality

Confidentiality, or privacy, of plaintexts, is the fundamental guarantee a symmetric encryption scheme aims to provide. We present three standard notions of confidentiality for symmetric encryption schemes. The first notion is IND-CPA (Indistinguishability under Chosen-Plaintext Attacks), that gives an adversary the opportunity to query plaintexts for encryption and receive back the corresponding ciphertexts. We say that the adversary has oracle access to the encryption algorithm. This notion first appeared in [17] (inspired by the public key equivalent notion [79]).

We present IND-CPA in terms of left-or-right indistinguishability. In this game, an adversary is given access to a left-or-right encryption oracle that encrypts one of two messages depending on a (uniformly random chosen) bit. To win the game, the adversary must return the bit b.

Definition 3 (IND-CPA).
Let $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme. Let* LR*, initialised by* INI*, be the algorithms defined in Figure 2.1. For any adversary* $\mathcal{A}$*, we define its* IND-CPA *advantage as:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{SE}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{INI} \,:\, \mathcal{A}^{\mathsf{LR}(\cdot,\cdot)} = b\right] - 1.$$

```
alg. INI
    $b \leftarrow\$ \{0,1\}$
    $\mathsf{k} \leftarrow \mathsf{Gen}$
    return

alg. $\mathsf{LR}(M_0, M_1)$
    if $|M_0| \neq |M_1|$
        return $\perp$
    $C \leftarrow \mathsf{Enc}_{\mathsf{k}}(M_b)$
    return $C$
```

Figure 2.1: Algorithms for defining IND-CPA.

The scheme SE *is* (η, R)-IND-CPA *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* IND-CPA *advantage is bounded by* η*.*

When examining the IND-CPA definition, it is not immediately obvious that it is a good definition, capturing any form of confidentiality. However, the notion can be shown to be equivalent to the notion of semantic security [80], which defines confidentiality in an intuitive way. We prefer the former notion because it is more user-friendly in proofs and easier to understand conceptually.

It is evident from the IND-CPA definition that a necessary condition on the symmetric encryption scheme is that the encryption algorithm must be randomised. This condition is required to avoid leakage when encryptions of the same plaintext are performed. In practice, deterministic encryption schemes are widely used, but with some additional external input that ensure randomisation of each encryption. Upcoming sections explore such constructions further.

An obvious way to strengthen IND-CPA further is to also give the adversary access to the decryption algorithm. The standard notion capturing this is named IND-CCA (Indistinguishability under Chosen-Ciphertext Attacks). In this notion, an adversary can, apart from being able to query plaintexts and receive back the corresponding ciphertexts, also query ciphertexts and receive back corresponding plaintexts. We say that the adversary has oracle access to both the encryption and decryption algorithms.

Definition 4 (IND-CCA)**.**
Let $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme. Let* LR *and* DEC*, both initialised by* INI*, be the algorithms defined in Figure 2.2. For any adversary* $\mathcal{A}$*, we defines its* IND-CCA *advantage as:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{SE}}(\mathcal{A}) = 2 \cdot \Pr\Big[\mathsf{INI} \,:\, \mathcal{A}^{\mathsf{LR}(\cdot,\cdot),\mathsf{DEC}(\cdot)} = b\Big] - 1.$$

```
alg. INI
    $b \leftarrow\!\$ \{0,1\}$
    $\mathsf{k} \leftarrow \mathsf{Gen}$
    $\mathcal{L}_{\mathsf{C}} = []$
    return
```

```
alg. LR($M_0, M_1$)
    if $|M_0| \neq |M_1|$
        return $\perp$
    $C \leftarrow \mathsf{Enc}_{\mathsf{k}}(M_b)$
    $\mathcal{L}_{\mathsf{C}}.\mathsf{append}(C)$
    return $C$
```

```
alg. DEC($C$)
    $M \leftarrow \mathsf{Dec}_{\mathsf{k}}(C)$
    if $C \in \mathcal{L}_{\mathsf{C}}$
        $M \leftarrow \perp$
    return $M$
```

Figure 2.2: Algorithms for defining IND-CCA.

The scheme SE *is* (η, R)-IND-CCA *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* IND-CCA *advantage is bounded by* η*.*

Note that the decryption oracle does not allow an adversary to query a ciphertext that was returned by the encryption oracle. If this would be allowed, the adversary could trivially win the game!

Random Bit Indistinguishability

There is a stronger notion of left-or-right indistinguishability known as *indistinguishability from random bits.* We present a CPA-style notion denoted by IND\$-CPA. In this notion, an adversary is tasked with distinguishing between real ciphertexts and strings of random bits, under the requirement that the random string of bits has the same length as the corresponding ciphertext. This notion first appeared in [123].

Definition 5 (IND\$-CPA)**.**
Let $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme. Let* ENC *and* \$*, both initialised by* INI*, be the algorithms defined in Figure 2.3. For any adversary* $\mathcal{A}$*, we define its* IND\$-CPA *advantage as:*

$$\mathsf{Adv}^{\mathsf{ind\$\text{-}cpa}}_{\mathsf{SE}}(\mathcal{A}) = \Pr\Big[\mathsf{INI} : \mathcal{A}^{\mathsf{ENC}(\cdot)} = 1\Big] - \Pr\Big[\mathsf{INI} : \mathcal{A}^{\$(\cdot)} = 1\Big].$$

The scheme SE *is* (η, R)-IND\$-CPA *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* IND\$-CPA *advantage is bounded by* η*.*

Since we assume length-regularity, IND\$-CPA implies IND-CPA (using a proof similar to Theorem 1 in [17]) with a security loss of only a factor of 2. The opposite is not true: there are schemes that meet IND-CPA but do not meet IND\$-CPA.

```
alg. INI
    $b \leftarrow\$ \{0,1\}$
    $\mathsf{k} \leftarrow \mathsf{Gen}$
    return

alg. ENC($M$)
    $C \leftarrow \mathsf{Enc}_{\mathsf{k}}(M)$
    return $C$

alg. \$($M$)
    $C \leftarrow \mathsf{Enc}_{\mathsf{k}}(M)$
    if $C = \perp$
        return $\perp$
    $R \leftarrow\$ \{0,1\}^{|C|}$
    return $R$
```

Figure 2.3: Algorithms for defining IND\$-CPA.

```
alg. INI
    $\mathsf{k} \leftarrow \mathsf{Gen}$
    $\mathsf{WIN} = \mathsf{false}$
    $\mathcal{L}_{\mathsf{M}} = []$
    return

alg. ENC($M$)
    $C \leftarrow \mathsf{Enc}_{\mathsf{k}}(M)$
    $\mathcal{L}_{\mathsf{M}}.\mathsf{append}(M)$
    return $C$

alg. VF($C$)
    $M \leftarrow \mathsf{Dec}_{\mathsf{k}}(C)$
    if $M \notin \mathcal{L}_{\mathsf{M}}$ and $M \neq \perp$
        $\mathsf{WIN} = \mathsf{true}$
    return $M$
```

Figure 2.4: Algorithms for defining INT-PTXT.

Authenticity

A second essential security property that a symmetric encryption scheme should aim to meet is authenticity of a plaintext. This property can be defined in terms of *integrity of plaintexts* as first defined in [22] and denoted by INT-PTXT.

Definition 6 (INT-PTXT)**.**
Let SE = (Gen, Enc, Dec) *be a symmetric encryption scheme. Let* ENC *and* VF*, both initialised by* INI*, be the algorithms defined in Figure 2.4. Let* FORGE *be the event that* WIN = true *after a query to* VF*. For any adversary* $\mathcal{A}$*, we define its* INT-PTXT *advantage as:*

$$\mathsf{Adv}^{\text{int-ptxt}}_{\mathsf{SE}}(\mathcal{A}) = \Pr\left[\mathsf{INI},\ \mathcal{A}^{\mathsf{ENC}(\cdot),\mathsf{VF}(\cdot)}\ :\ \mathsf{FORGE}\right].$$

The scheme SE *is* (η, R)-INT-PTXT *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* INT-PTXT *advantage is bounded by* η*.*

An adversary wins the game by querying a valid ciphertext to the algorithm VF (i.e. a ciphertext that decrypts to a plaintext that is not the error symbol), and that does not decrypt to a plaintext that has previously been queried to the algorithm ENC.

A stronger notion, also identified by Bellare and Namprempre [22] is *integrity of ciphertexts*, INT-CTXT.

```
alg. $\mathsf{INI}$
    $\mathsf{k} \leftarrow \mathsf{Gen}$
    $\mathsf{WIN} = \mathsf{false}$
    $\mathcal{L}_\mathsf{C} = []$
    return

alg. $\mathsf{ENC}(M)$
    $C \leftarrow \mathsf{Enc}_\mathsf{k}(M)$
    $\mathcal{L}_\mathsf{C}.\mathsf{append}(C)$
    return $C$

alg. $\mathsf{VF}(C)$
    $M \leftarrow \mathsf{Dec}_\mathsf{k}(C)$
    if $C \notin \mathcal{L}_\mathsf{C}$ and $M \neq \perp$
        $\mathsf{WIN} = \mathsf{true}$
    return $M$
```

Figure 2.5: Algorithms for defining INT-CTXT.

Definition 7 (INT-CTXT).
Let $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme. Let* ENC *and* VF*, both initialised by* INI*, be the algorithms defined in Figure 2.5. Let* FORGE *be the event that* $\mathsf{WIN} = \mathsf{true}$ *after a query to* VF*. For any adversary* $\mathcal{A}$*, we defines its* INT-CTXT *advantage as:*

$$\mathsf{Adv}^{\mathsf{int\text{-}ctxt}}_{\mathsf{SE}}(\mathcal{A}) = \Pr\left[\mathsf{INI},\ \mathcal{A}^{\mathsf{ENC}(\cdot),\mathsf{VF}(\cdot)}\ :\ \mathsf{FORGE}\right].$$

The scheme SE *is* (η, R)-INT-CTXT *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* INT-CTXT *advantage is bounded by* η*.*

The INT-CTXT notion is very similar to INT-PTXT. However, instead of requiring that the plaintext, produced by decrypting the queried ciphertext, must not have been queried to ENC, it is required that the queried ciphertext has not been returned by ENC and the queried ciphertext should decrypt to some plaintext different from the error symbol.

While the two notions look similar, INT-CTXT is a strictly stronger notion than INT-PTXT. This can be seen by observing that if C is a ciphertext queried to VF, and the resulting plaintext M has *not* been queried to ENC, then C cannot have been returned by ENC without violating correctness of the symmetric encryption scheme. Hence, we can use an adversary for INT-PTXT to construct an adversary for INT-CTXT. For the strict part, consider an INT-PTXT secure scheme. Modify the decryption algorithm such that it ignores any excess ciphertext (identifiable in an appropriate way). Then the modified scheme is still INT-PTXT secure but definitely not INT-CTXT secure.

We note that the authenticity property is also commonly referred to by the name *integrity*. We use both names interchangeably in this thesis.

Authenticated Encryption

An authenticated encryption scheme aims to met the notions of confidentiality and authenticity simultaneously and follows the same syntax as defined in Definition 2. Bellare and Namprempre [22] identified $\mathsf{IND\text{-}CCA}$ and $\mathsf{INT\text{-}PTXT}$ as being the suitable notions for confidentiality and authenticity, respectively. When considering whether a scheme is an authenticated encryption scheme, Bellare and Namprempre proved a very useful result (which we will not prove here), given informally below. Note, Theorem 1 is not true in general if we allow the symmetric encryption scheme to have more than one error type, cf. [37].

Theorem 1 ($\mathsf{IND\text{-}CPA}$ and $\mathsf{INT\text{-}CTXT}$ implies $\mathsf{IND\text{-}CCA}$)**.**
Let SE *be a symmetric encryption scheme. For any* $\mathsf{IND\text{-}CCA}$ *adversary* $\mathcal{A}_{\mathsf{cca}}$ *there exist adversaries* $\mathcal{A}_{\mathsf{cpa}}$ *and* $\mathcal{A}_{\mathsf{ctxt}}$*, consuming "similar" resources to* $\mathcal{A}_{\mathsf{cca}}$*, such that:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{SE}}(\mathcal{A}_{\mathsf{cca}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{int\text{-}ctxt}}_{\mathsf{SE}}(\mathcal{A}_{\mathsf{ctxt}}) + \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{SE}}(\mathcal{A}_{\mathsf{cpa}}).$$

Hence, given a symmetric encryption scheme SE, if we can show that SE meets $\mathsf{IND\text{-}CPA}$ and $\mathsf{INT\text{-}CTXT}$, we have shown that SE meets $\mathsf{INT\text{-}PTXT}$ and $\mathsf{IND\text{-}CCA}$. Bellare and Namprempre [22] show that the converse of Theorem 1 it not true, by showing that $\mathsf{INT\text{-}PTXT}$ is not guaranteed by $\mathsf{IND\text{-}CCA}$.

It is possible to capture the desired security properties (our definition below actually capture the slightly stronger combination of $\mathsf{IND\$\text{-}CPA}$ and $\mathsf{INT\text{-}CTXT}$) of an authenticated encryption scheme in an all-in-one notion first proposed by Rogaway [122]. We will denote this notion by AE. In Section 2.3.5, we will explore this type of notion more when defining IV-based and nonce-based symmetric encryption schemes.

Definition 8 (AE)**.**
Let $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme. Let* ENC*,* DEC*,* $\mathsf{\$}$ *and* Err*, all initialised by* INI*, be the algorithms defined in Fig. 2.6. For any adversary* $\mathcal{A}$*, we define its* AE *advantage as:*

$$\mathsf{Adv}^{\mathsf{AE}}_{\mathsf{nSE}}(\mathcal{A}) = \Pr\left[\mathsf{INI}\ :\ \mathcal{A}^{\mathsf{ENC}(\cdot),\mathsf{DEC}(\cdot)} = 1\right] - \Pr\left[\mathsf{INI}\ :\ \mathcal{A}^{\mathsf{\$}(\cdot),\mathsf{Err}(\cdot)} = 1\right].$$

The scheme SE *is* (η, R)*-*AE *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* AE *advantage is bounded by* η*.*

We call a symmetric encryption scheme satisfying Definition 8 an *authenticated symmetric encryption scheme*, sometimes omitting the word symmetric.

```
alg. INI
  $\mathsf{k} \leftarrow \mathsf{Gen}$
  $\mathcal{L}_\mathsf{C} = []$
  return

alg. $(M)
  $C \leftarrow \mathsf{Enc}_\mathsf{k}(M)$
  if $C = \perp$
    return $\perp$
  $R \leftarrow\$ \{0,1\}^{|C|}$
  return $R$

alg. ENC(M)
  $C \leftarrow \mathsf{Enc}_\mathsf{k}(M)$
  $\mathcal{L}_\mathsf{C}.\mathsf{append}(C)$
  return $C$

alg. Err(C)
  return $\perp$

alg. DEC(C)
  if $C \in \mathcal{L}_\mathsf{C}$
    return $\perp$
  $M \leftarrow \mathsf{Dec}_\mathsf{k}(C)$
  return $M$
```

Figure 2.6: Algorithms for defining AE.

In the AE security game, the adversary is given oracle access to one of two pairs of algorithms. The first pair of algorithms consists of the encryption and decryption algorithms. In the second pair of algorithms, one algorithm outputs random strings (of the same length as the corresponding ciphertexts) while the second algorithm always outputs the error symbol $\perp$. Informally, the definition measures the ability of an adversary to distinguish an encryption-decryption oracle pair from a pair of oracles that return random bits and errors.

It is trivial to see that if a symmetric encryption scheme meets AE then it also meets IND\$-CPA, by simply ignoring the DEC and Err oracle access. On the same lines, consider an adversary that wins the INT-CTXT game. The adversary must first satisfy the condition that $C \notin \mathcal{L}_\mathsf{C}$. Secondly, the adversary must also ensure that the decryption algorithm does not return an error. But if an adversary can satisfy these two conditions, the adversary can also win the AE game, because a distinguisher can distinguish between the output from algorithms DEC and Err. Together with Theorem 1 this shows that AE also implies IND-CCA.

2.3.5 IV-based and Nonce-based Symmetric Encryption

In the previous section, we defined probabilistic symmetric encryption schemes. However, in practice, symmetric encryption schemes are often not built using internal randomness. They instead rely on externally provided values and let the encryption algorithm be deterministic. For example, widely used symmetric encryption schemes such as AES-CBC and AES-GCM [107] rely on external values (although, the former could be defined as a probabilistic scheme). The former uses a uniformly random chosen *initialisation vector* (IV) and the latter utilises a *nonce*.

Syntax

Below we present definitions for both IV-based and nonce-based symmetric encryptions schemes, with the presentation inspired by Namprempre et al. [112].

Definition 9 (IV-based symmetric encryption scheme)**.**
An IV-based symmetric encryption scheme $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with key space* $\mathcal{K} \subseteq \{0,1\}^*$*, plaintext space* $\mathcal{M} \subseteq \{0,1\}^*$*, ciphertext space* $\mathcal{C} \subseteq \{0,1\}^*$*, initialisation vector space* $\mathcal{I} \subseteq \{0,1\}^*$ *and error symbol* $\perp$ *(*$\perp \notin \mathcal{M} \cup \mathcal{C}$*) is specified by three algorithms:*

- *A randomised key generation algorithm* Gen *that outputs a key* $\mathsf{k} \in \mathcal{K}$*. We write* $\mathsf{k} \leftarrow \mathsf{Gen}$*.*
- *A deterministic encryption algorithm* Enc *that takes as input a key* k*, a plaintext* $m \in \mathcal{M}$ *and initialisation vector* $iv \in \mathcal{I}$*, and outputs a ciphertext* $c \in \mathcal{C} \cup \{\perp\}$*. We write* $c \leftarrow \mathsf{Enc}_{\mathsf{k}}(m, iv)$*.*
- *A deterministic decryption algorithm* Dec *that takes as input a key* k*, a ciphertext* $c \in \mathcal{C}$ *and initialisation vector* $iv \in \mathcal{I}$*, and outputs a plaintext* $m \in \mathcal{M} \cup \{\perp\}$*. We write* $m \leftarrow \mathsf{Dec}_{\mathsf{k}}(c, iv)$*.*

The IV iv that is provided as input to the encryption algorithm should be chosen uniformly at random from the set of all possible IV values $\mathcal{I}$. This will be enforced by the encryption oracle algorithm in the game defining the desired security properties of an IV-based symmetric encryption scheme. In addition, the encryption oracle algorithm will surface the initialisation vector, which reflects the practical requirement that a receiver of the ciphertext requires the IV for decryption.

Furthermore, we require that $\mathcal{M}$ contains at last two strings and that if $\mathcal{M}$ contains a string of length l, it contains all strings of length l. Finally, we assume that all IV-based symmetric encryption schemes are length-regular. That is, if $c \leftarrow \mathsf{Enc}_{\mathsf{k}}(m, iv)$ and $c \neq \perp$, then the length of c is a function of m and iv.

For lists $\mathcal{L}_m = [m_1, m_2, \ldots, m_l] \in [\mathcal{M}]^l$ and $\mathcal{L}_{iv} = [iv_1, iv_2, \ldots, iv_l] \in [\mathcal{I}]^l$, we write $\mathcal{L}_c \leftarrow \mathsf{Enc}_{\mathsf{k}}(\mathcal{L}_m, \mathcal{L}_{iv})$ where $\mathcal{L}_c = [c_1, c_2, \ldots, c_l]$ and $c_1 \leftarrow \mathsf{Enc}_{\mathsf{k}}(m_1, iv_1)$, $c_2 \leftarrow$

$\mathsf{Enc_k}(m_2, iv_2)$, …, $c_l \leftarrow \mathsf{Enc_k}(m_l, iv_l)$. For a list $\mathcal{L}_c = [c_1, c_2, \ldots, c_l] \in [\{0,1\}^*]^l$, we likewise write $\mathcal{L}_m \leftarrow \mathsf{Dec_k}(\mathcal{L}_c, \mathcal{L}_{iv})$ where $\mathcal{L}_m = [m_1, m_2, \ldots, m_l]$ and $m_1 \leftarrow \mathsf{Dec_k}(c_1, iv_1)$, $m_2 \leftarrow \mathsf{Dec_k}(c_2, iv_2), \ldots$, $m_l \leftarrow \mathsf{Dec_k}(c_l, iv_l)$.

Definition 10 (Nonce-based symmetric encryption scheme).
A nonce-based symmetric encryption scheme $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with key space* $\mathcal{K} \subseteq \{0,1\}^*$*, plaintext space* $\mathcal{M} \subseteq \{0,1\}^*$*, ciphertext space* $\mathcal{C} \subseteq \{0,1\}^*$*, nonce space* $\mathcal{N} \subseteq \{0,1\}^*$*, additional data space* $\mathcal{A} \subseteq \{0,1\}^*$ *and error symbol* $\perp$ *(*$\perp \notin \mathcal{M} \cup \mathcal{C}$*) is specified by three algorithms:*

- *A randomised key generation algorithm* Gen *that outputs a key* k*. We write* $\mathsf{k} \leftarrow \mathsf{Gen}$*.*
- *A deterministic encryption algorithm* Enc *that takes as input a key* k*, a plaintext* $m \in \mathcal{M}$*, a nonce* $n \in \mathcal{N}$ *and additional data* $a \in \mathcal{A}$*, and outputs a ciphertext* $c \in \mathcal{C} \cup \{\perp\}$*. We write* $c \leftarrow \mathsf{Enc_k}(m, n, a)$*.*
- *A deterministic decryption algorithm* Dec *that takes as input a key* k*, a ciphertext* $c \in \mathcal{C}$*, a nonce* $n \in \mathcal{N}$ *and additional data* $a \in \mathcal{A}$*, and outputs a plaintext* $m \in \mathcal{M} \cup \{\perp\}$*. We write* $m \leftarrow \mathsf{Dec_k}(c, n, a)$*.*

We will sometimes omit the additional data from the notation when this is not required. The nonce should be handled with care, because, as it will become apparent soon, the security of nonce-based schemes relies on the nonce to never repeat during calls to encryption. Since the nonce is an external input, it is up to the consumer to honour this requirement. This has created a basis for misuse since encryption is often used in complex protocols or environments where ensuring non-repetition of nonces can be challenging to achieve [35, 96, 40, 58, 126, 131].

Furthermore, we require that $\mathcal{M}$ contains at last two strings and that if $\mathcal{M}$ contains a string of length l, it contains all strings of length l. Finally, all nonce-based symmetric encryption schemes are assumed to be length-regular. That is, if $c \leftarrow \mathsf{Enc_k}(m, n, a)$ and $c \neq \perp$, then the length of c is only a function of m, n and a.

For lists $\mathcal{L}_m = [m_1, m_2, \ldots, m_l] \in [\mathcal{M}]^l$, $\mathcal{L}_n = [n_1, n_2, \ldots, n_l] \in [\mathcal{N}]^l$ and $\mathcal{L}_a = [a_1, a_2, \ldots, a_l] \in [\mathcal{A}]^l$, we write $\mathcal{L}_c \leftarrow \mathsf{Enc_k}(\mathcal{L}_m, \mathcal{L}_n, \mathcal{L}_a)$ where $\mathcal{L}_c = [c_1, c_2, \ldots, c_l]$ and $c_1 \leftarrow \mathsf{Enc_k}(m_1, n_1, a_1)$, $c_2 \leftarrow \mathsf{Enc_k}(m_2, n_2, a_2), \ldots$, $c_l \leftarrow \mathsf{Enc_k}(m_l, n_l, a_l)$. For a list $\mathcal{L}_c = [c_1, c_2, \ldots, c_l] \in [\{0,1\}^*]^l$, we likewise write $\mathcal{L}_m \leftarrow \mathsf{Dec_k}(\mathcal{L}_c, \mathcal{L}_n, \mathcal{L}_a)$ where $\mathcal{L}_m = [m_1, m_2, \ldots, m_l]$ and $m_1 \leftarrow \mathsf{Dec_k}(c_1, n_1, a_1)$, $m_2 \leftarrow \mathsf{Dec_k}(c_2, n_2, a_2), \ldots$, $m_l \leftarrow \mathsf{Dec_k}(c_l, n_l, a_l)$.

Correctness

As was the case in Section 2.3.4, we assume that all IV-based and nonce-based symmetric encryption schemes are perfectly correct. In the IV case: for all keys $\mathsf{k} \in \mathcal{K}$, that have a non-zero probability of being output by Gen, all $m \in \mathcal{M}$ and all $iv \in \mathcal{I}$ the following is true with probability one:

$$\text{If } c \leftarrow \mathsf{Enc}_{\mathsf{k}}(m, iv) \text{ and } c \neq \bot, \text{then } m \leftarrow \mathsf{Dec}_{\mathsf{k}}(c, iv).$$

For the nonce case: for all keys $\mathsf{k} \in \mathcal{K}$, that have non-zero probability of being output by Gen, all $m \in \mathcal{M}$, all $n \in \mathcal{N}$ and all $a \in \mathcal{A}$ the following is true with probability one:

$$\text{If } c \leftarrow \mathsf{Enc}_{\mathsf{k}}(m, n, a) \text{ and } c \neq \bot, \text{then } m \leftarrow \mathsf{Dec}_{\mathsf{k}}(c, n, a).$$

Security Notions

Security notions for IV-based and nonce-based symmetric encryption schemes are similar in style to Definition 8, taking into account IV's/nonces and ruling out any additional trivial wins. These security notions were first presented by Namprempre et al. [112]. We replicate these notions with a slightly changed notation.

In the security notions below there is a noteworthy distinction in the captured security properties between IV-based schemes and nonce-based schemes. In the latter case, we aim to capture authenticated encryption properties, providing both confidentiality and authenticity. In the former case, we only aim to capture confidentiality. An IV-based scheme must be paired with another mechanism to yield a construction that meets both properties.

Definition 11 (ivE)**.**
Let $\mathsf{ivSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be an IV-based symmetric encryption scheme. Let* ENC, $\$$*, both initialised by* INI*, be the algorithms defined in Figure 2.7. For any adversary* $\mathcal{A}$*, we define its* ivE *advantage as:*

$$\mathsf{Adv}^{\mathsf{ivE}}_{\mathsf{ivSE}}(\mathcal{A}) = \Pr\left[\mathsf{INI} : \mathcal{A}^{\mathsf{ENC}(\cdot)} = 1\right] - \Pr\left[\mathsf{INI} : \mathcal{A}^{\$(\cdot)} = 1\right].$$

The scheme ivSE *is* (η, R)-ivE *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* ivE *advantage is bounded by* η*.*

```
alg. $\mathsf{INI}$
$\mathsf{k} \leftarrow \mathsf{Gen}$
return

alg. $\mathsf{ENC}(M)$
$IV \leftarrow\!\$\ \mathcal{I}$
$C \leftarrow \mathsf{Enc}_{\mathsf{k}}(M, IV)$
return $(IV, C)$

alg. $\$(M)$
$IV \leftarrow\!\$\ \mathcal{I}$
$C \leftarrow \mathsf{Enc}_{\mathsf{k}}(M, IV)$
if $C = \perp$
    return $\perp$
$R \leftarrow\!\$\ \{0,1\}^{|C|}$
return $(IV, R)$
```

Figure 2.7: Algorithms for defining ivE.

We next present the security notion for nonce-based symmetric encryption schemes. This notion is very similar to the IV-based notion, with the caveat that an adversary must be nonce-respecting, meaning that the adversary must never use the same nonce twice in queries to the encryption oracles ENC and $\$$.

Definition 12 (nAE)**.**
Let $\mathsf{nSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a nonce-based symmetric encryption scheme. Let* ENC*,* DEC*,* $\$$ *and* Err*, all being initialised by* INI*, be the algorithms defined in Figure 2.8. For any adversary* $\mathcal{A}$*, we define its* nAE *advantage as:*

$$\mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}) = \Pr\Big[\mathsf{INI} \,:\, \mathcal{A}^{\mathsf{ENC}(\cdot,\cdot,\cdot),\mathsf{DEC}(\cdot,\cdot,\cdot)} = 1\Big] - \Pr\Big[\mathsf{INI} \,:\, \mathcal{A}^{\$(\cdot,\cdot,\cdot),\mathsf{Err}(\cdot,\cdot,\cdot)} = 1\Big].$$

The scheme nSE *is* (η, R)*-*nAE *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* nAE *advantage is bounded by* η*.*

The security game for the nAE notion restricts the adversary to not repeat nonces and to not query, to the decryption oracle, ciphertexts that were returned by the encryption oracle ENC.

The formulation of the security notion for a nonce-based symmetric encryption scheme is similar to the security notion definition given for an authenticated symmetric encryption scheme in Definition 8. We will, therefore, also refer to a nonce-based symmetric encryption scheme as a *nonce-based AE scheme.* We use the two terminologies interchangeably.

2.3.6 Block Ciphers

Consider a function $\mathsf{F}\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, where k is the *key length/size* and n is the *block length/size.* For each $\mathsf{k} \in \{0,1\}^k$, we write $\mathsf{F}_{\mathsf{k}}\colon \{0,1\}^n \to \{0,1\}^n$

```
alg. INI
k ← Gen
$\mathcal{L}_\mathsf{N} = []$
$\mathcal{L}_\mathsf{C} = []$
return
```

```
alg. ENC$(M, N, A)$
if $N \in \mathcal{L}_\mathsf{N}$
  return $\perp$
$C \leftarrow \mathsf{Enc}_\mathsf{k}(M, N, A)$
$\mathcal{L}_\mathsf{N}$.append$(N)$
$\mathcal{L}_\mathsf{C}$.append$(C)$
return $C$
```

```
alg. DEC$(C, N, A)$
if $C \in \mathcal{L}_\mathsf{C}$
  return $\perp$
$M \leftarrow \mathsf{Dec}_\mathsf{k}(C, N, A)$
return $M$
```

```
alg. $\$(M, N, A)$
if $N \in \mathcal{L}_\mathsf{N}$
  return $\perp$
$C \leftarrow \mathsf{Enc}_\mathsf{k}(M, N, A)$
$\mathcal{L}_\mathsf{N}$.append$(N)$
if $C = \perp$
  return $\perp$
$R \leftarrow_\$ \{0,1\}^{|C|}$
return $R$
```

```
alg. Err$(C, N, A)$
return $\perp$
```

Figure 2.8: Algorithms for defining nAE.

defined as $\mathsf{F}_\mathsf{k}(x) = \mathsf{F}(\mathsf{k}, x)$. If F is a permutation, we call F a block cipher. Since the block cipher F is a permutation, there exists an inverse, denoted by F^{-1}, such that $\mathsf{F}_\mathsf{k}^{-1}(\mathsf{F}_\mathsf{k}(x)) = x$. As mentioned above, we generally want a block cipher to be a PRF and, in some cases, a PRP.

Given a block cipher F, we can define an IV-based symmetric encryption scheme that uses the CBC (Cipher Block Chaining) mode of operation.

Definition 13 (CBC IV-based symmetric encryption scheme)**.**
Let $\mathsf{F}\colon \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ *be a block cipher. Define the IV-based symmetric encryption scheme* $\mathsf{ivSE_{CBC}} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$*, with key space* $\mathcal{K} = \{0,1\}^k$*, message space* $\mathcal{M} = \{x \in \{0,1\}^* \mid |x| \bmod n = 0\}$*, ciphertext space* $\mathcal{C} = \{x \in \{0,1\}^* \mid |x| \bmod n = 0\}$ *and IV space* $\mathcal{I} = \{0,1\}^n$*, as follows:*

Gen**:** *Sample* $\mathsf{k} \leftarrow_\$ \{0,1\}^k$ *and return* k*.*

$\mathsf{Enc}_\mathsf{k}(m, iv)$**:** *Write* $m = m_1 \parallel m_2 \parallel \cdots \parallel m_l$ *for* $m_i \in \{0,1\}^n$ *and* $c_0 = iv$*. Let* $c_i = \mathsf{F}_\mathsf{k}(c_{i-1} \oplus m_i)$ *(*$i = 1, 2, \ldots, l$*) and* $c = c_1 \parallel c_2 \parallel \cdots \parallel c_l$*. Return* c*.*

$\mathsf{Dec}_\mathsf{k}(c, iv)$**:** *Write* $c = c_1 \parallel \cdots \parallel c_l$ *for* $c_i \in \{0,1\}^n$ *and let* $c_0 = iv$*. Let* $m_i = \mathsf{F}_\mathsf{k}^{-1}(c_i) \oplus c_{i-1}$ *(*$i = 1, 2, \ldots, l$*) and* $m = m_1 \parallel m_2 \parallel \cdots \parallel m_l$*. Return* m*.*

If $\mathsf{ivSE_{CBC}}$ uses uniformly random IVs, then Bellare et al. [17] proved that it meets

ivE if F is a PRP. However, this result is void if the IVs are not uniformly random. In Chapter 5, we present several attacks against CBC-mode based schemes in SSH.

2.3.7 Message Authentication

In this section, we give a definition of message authentication schemes and present security notions that capture the desired security properties of such schemes. A message authentication scheme enables two remote peers to verify that the message they receive did, in fact, originate from the other peer. This requires them to have a shared key, only known to them.

Syntax

Definition 14 (Message authentication scheme).
A message authentication scheme $\mathsf{MA} = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$ *with key space* $\mathcal{K} \subseteq \{0,1\}^*$ *and message space* $\mathcal{M} \subseteq \{0,1\}^*$ *is specified by three algorithms:*

- *A randomised key generation algorithm* Gen *that outputs a key* $\mathsf{k} \in \mathcal{K}$*. We write* $\mathsf{k} \leftarrow \mathsf{Gen}$*.*
- *A deterministic tagging algorithm* Tag *that takes as input a key* k *and a message* $m \in \mathcal{M}$*, and outputs a tag* $\tau \in \{0,1\}^*$*. We write* $\tau \leftarrow \mathsf{Tag}_{\mathsf{k}}(m)$*.*
- *A deterministic verification algorithm* Ver *that takes as input a key* k*, a message* $m \in \mathcal{M}$ *and an expected tag* τ_{expected}*, and outputs a bit* $v \in \{0,1\}$*. We write* $v \leftarrow \mathsf{Ver}_{\mathsf{k}}(m, \tau_{\mathsf{expected}})$*. A tag* τ_{expected} *is valid for message* m *if and only if* $v = 1$*.*

There is an important subclass of message authentication schemes called *Message Authentication Codes* (*MAC*). The subclass consists of all message authentication schemes for which the verification algorithm simply computes the tag on the message using the tagging algorithm and then checks the output against the expected tag, see Figure 2.9.

Definition 15 (Message authentication code (MAC)).
A message authentication code $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac})$ *with key space* $\mathcal{K} \subseteq \{0,1\}^*$ *and message space* $\mathcal{M} \subseteq \{0,1\}^*$ *is specified by two algorithms:*

```
alg. $\mathsf{Ver}_{\mathsf{k}}(m, \tau_{\mathsf{expected}})$
$\tau \leftarrow \mathsf{Tag}_{\mathsf{k}}(m)$
if $\tau = \tau_{\mathsf{expected}}$
  return 1
return 0
```

Figure 2.9: MAC verification algorithm.

- *A randomised key generation algorithm* Gen *that outputs a key* k*. We write* $\mathsf{k} \leftarrow \mathsf{Gen}$*.*
- *A deterministic tagging algorithm* Mac *that takes as input a key* k *and a message* $m \in \mathcal{M}$*, and outputs a tag* $\tau \in \{0,1\}^*$*. We write* $\tau \leftarrow \mathsf{Mac}_{\mathsf{k}}(m)$*.*

The verification algorithm is fully specified by the tagging algorithm and is depicted in Figure 2.9.

Unfortunately, there are several serious pitfalls associated with verifying a MAC tag. For example, not verifying the tag in constant-time (i.e. in time independent of the position of first bit in tag that does not match the received tag), can lead to vulnerabilities. Such technicalities can easily be overlooked, even by experts, as exemplified with Google's cryptographic toolkit Keyczar (now deprecated) [100], a very recent 2019 CVE on Apache Tapestry [127] and the Xbox 360 timing attack in 2007 [118]. Furthermore, making the tag verification dependent on code-paths and, simultaneously, not unifying observable output after a tag verification can also lead to vulnerabilities. We will see several examples of such a vulnerability in Chapter 5.

Correctness

We require the following correctness condition to be true. For any keys $\mathsf{k} \in \mathcal{K}$, that can be output by Gen, and any message $m \in \mathcal{M}$ the following holds with probability 1:

$$\text{If } \tau \leftarrow \mathsf{Tag}_{\mathsf{k}}(m), \text{ then } \mathsf{Ver}_{\mathsf{k}}(m, \tau) = 1.$$

In addition, given a tag τ, we require the *tag length* $|\tau|$ to be "constant" over all messages. That is, there exists an integer $\ell_{\mathsf{tag}} \geq 1$ such that for any key $\mathsf{k} \in \mathcal{K}$, that can be output by Gen, and any message $m \in \mathcal{M}$ the following holds with probability

```
alg. INI
    $k \leftarrow \mathsf{Gen}$
    $\mathsf{WIN} = \mathsf{false}$
    $\mathcal{L}_{\mathsf{M}} = []$
    return
```

```
alg. TAG($M$)
    $\tau \leftarrow \mathsf{Tag}_{\mathsf{k}}(M)$
    $\mathcal{L}_{\mathsf{M}}.\mathsf{append}(M)$
    return $\tau$
```

```
alg. VER($M, \tau_{\mathsf{expected}}$)
    $v \leftarrow \mathsf{Ver}_{\mathsf{k}}(M, \tau_{\mathsf{expected}})$
    if $v = 1$ and $M \notin \mathcal{L}_{\mathsf{M}}$
        $\mathsf{WIN} = \mathsf{true}$
    return $v$
```

Figure 2.10: Algorithms for defining UF-CMA.

1:

$$\text{If } \tau \leftarrow \mathsf{Tag}_{\mathsf{k}}(m), \text{ then } |\tau| = \ell_{\mathsf{tag}}.$$

This condition is akin to the length-regular condition for symmetric encryption schemes.

We henceforth assume that any message authentication scheme encountered in this thesis satisfy the two conditions above.

Security Notion

The standard security property that message authentication schemes aim to meet is *existential unforgeability under chosen message attacks*, or in short, UF-CMA. In this notion, an adversary must produce a message and a tag that verify (with respect to Ver). The adversary is given oracle access to the tagging algorithm but is restricted not to reuse a message that was already queried to the tagging oracle TAG.

Definition 16 (UF-CMA).
Let $\mathsf{MA} = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Ver})$ *be a message authentication scheme. Let* TAG *and* VER*, both initialised by* INI*, be the algorithms defined in Figure 2.10. Let* FORGE *be the event that* $\mathsf{WIN} = \mathsf{true}$ *after a query to* VER*. For any adversary* $\mathcal{A}$*, we define its* UF-CMA *advantage as:*

$$\mathsf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathsf{MA}}(\mathcal{A}) = \Pr\Big[\mathsf{INI},\ \mathcal{A}^{\mathsf{TAG}(\cdot),\mathsf{VER}(\cdot)}\ :\ \mathsf{FORGE}\Big]\,.$$

The scheme MA *is* (η, R)-UF-CMA *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* UF-CMA *advantage is bounded by* η*.*

There exists a stronger notion than UF-CMA, that allows an adversary to reuse a message for its forgery. This notion is named *strong unforgeability under chosen*

```
alg. INI
k ← Gen
WIN = false
$\mathcal{L}_{\mathsf{M}} = []$
return
```

```
alg. TAG($M$)
$\tau \leftarrow \mathsf{Tag}_{\mathsf{k}}(M)$
$\mathcal{L}_{\mathsf{M}}.\mathsf{append}((M, \tau))$
return $\tau$
```

```
alg. VER($M, \tau_{\mathsf{expected}}$)
$v \leftarrow \mathsf{Ver}_{\mathsf{k}}(M, \tau_{\mathsf{expected}})$
if $v = 1$ and $(M, \tau_{\mathsf{expected}}) \notin \mathcal{L}_{\mathsf{M}}$
  WIN = true
return $v$
```

Figure 2.11: Algorithms for defining SUF-CMA.

message attacks and denoted by SUF-CMA. This notion only verifies that a message-tag pair has not previously been produced by a query to the tagging oracle. This allows an adversary to win by only producing a new tag for a message.

Definition 17 (SUF-CMA)**.**
Let MA = (Gen, Tag, Ver) *be a message authentication scheme. Let* TAG *and* VER*, both initialised by* INI*, be the algorithms defined in Figure 2.11. Let* FORGE *be the event that* WIN = true *after a query to* VER*. For any adversary $\mathcal{A}$, we define its* SUF-CMA *advantage as:*

$$\mathsf{Adv}^{\mathsf{suf\text{-}cma}}_{\mathsf{MA}}(\mathcal{A}) = \Pr\Big[\mathsf{INI},\ \mathcal{A}^{\mathsf{TAG}(\cdot),\mathsf{VER}(\cdot)}\ :\ \mathsf{FORGE}\Big].$$

The scheme MA *is* (η, R)-SUF-CMA *secure, if for any adversary $\mathcal{A}$ with resources at most* R*, its* SUF-CMA *advantage is bounded by η.*

Clearly, if an adversary can produce a forgery in the UF-CMA sense, then that forgery is also valid in the SUF-CMA sense (the message part of a message-tag pair has not been queried to the tagging oracle). That is, SUF-CMA implies UF-CMA. The former is, in fact, a strictly stronger notion. This can be shown with a similar argument used to argue for separability between the INT-CTXT and INT-PTXT notions. On the other hand, the two notions have equivalent strength when we restrict to the subclass of MACs.

Consider a function $\mathsf{F}\colon \mathcal{X} \to \mathcal{Y}$ with all elements of $\mathcal{Y}$ having the same bit-length. Suppose we define F to be the tagging algorithm and verify tags using the MAC verification method described earlier. Then, it is possible to show that if F is PRF secure then F is also UF-CMA secure with only a loss of $1/2^{|\mathcal{Y}|}$ in the reduction, requiring the co-domain $\mathcal{Y}$ to be sufficiently large for the reduction to make sense. We will not go into details, but refer the reader to [77, 19].

2.4 SSH

This thesis has a big focus on SSH. In this section, we will therefore provide some background on SSH, explain its Binary Packet Protocol and describe two of the most popular implementations of SSH.

SSH (Secure SHell) is a general-purpose secure network communication protocol and standardised in a series of five RFCs: RFC4250 [101], RFC4251 [134], RFC4252 [132], RFC4253 [135], RFC4254 [133]. There are two main versions of SSH: version 1 and version 2. This thesis will only be concerned with version 2 and all references to SSH will be for this specific version.[2] SSH facilitates secure remote login and secure use of a mixture of network services.

The SSH protocol is very widely used, and all major operating systems come with built in tools to use SSH. It is the dominant protocol used for automation and machine management through tools such as *ansible* [48], and is one of the protocols that can be used as the transport layer in *Git* [128].

SSH is often misconstrued as being similar to the network protocol TLS. This is true when considering the two protocols at a high level; they are both used to securely transfer data on a network. However, on a technical level they are significantly different. Firstly, SSH relies on the Trust On First Use (TOFU) trust model, where a client sets up the trust relationship upon first connection to a remote endpoint. Secondly, while TLS is solely a transport protocol, SSH defines a service layer going beyond transport (this layer is coined the *connection protocol*, runs on top of the transport layer protocol [135] and user authentication protocol [132], and is defined in [133]). For example, a client can request a shell through an SSH session that makes it possible to connect to execute commands on a remote machine. This is a widely used functionality for remote machine administration. Finally, the packet format in the two protocols is significantly different both in layout and how cryptography is applied to them.

[2] SSH version 1 is not secure. For example, version 1 uses CRC-32 (32-bit Cyclic Redundancy Check) integrity checks to provide authentication.

2.4.1 SSH Implementations

In Chapter 4, we will present a detailed discussion of several scans revealing a number of interesting statistics on SSH servers reachable on the Internet. Among other things, we highlight which implementations are most prevalent on the server-side and their preferred choice of SSH encryption scheme. As we will see, the SSH landscape is dominated by only two implementations: *OpenSSH* and *Dropbear*. Below we give a short introduction to both.

OpenSSH

OpenSSH [66] has been one of the most prominent SSH implementations since its inception in September 26, 1999 [50]. It was first released for public use in the operating system OpenBSD on December 1, 1999 [50]. Early OpenSSH developers did not start from scratch, but forked from the SSH implementation OSSH, written by Björn Grönvall, which, itself, was a fork of Tatu Ylönen's SSH implementation (the author of the original SSH protocol). Ylönen later formed the company *SSH Communications Security Corp* to commercialise his SSH efforts. This later led to a naming controversy in the community [106].

OpenSSH is a complete client and server-side implementation of the SSH version 2 protocol. However, it also extends the protocol with additional features such as pluggable authentication modules (PAM), certificate authentication, and an expanded set of available cryptographic algorithms. In addition, OpenSSH includes a suite of command-line tools such as SCP and SFTP that can be used to perform secure file transfer. OpenSSH used to contain support for the SSH version 1 protocol, but support was completely removed from version 7.6 [71] onwards. The newest version of OpenSSH is version 8.1 [74] (at the time of writing).

OpenSSH can use both *LibreSSL* [49] or *OpenSSL* [46] as its cryptographic provider, natively targeting the former. As a special feature, OpenSSH can be built without linking to an external cryptographic provider and falls back to its own, low-level cryptographic implementations. However, doing this disables a number of OpenSSH's encryption schemes, e.g. SSH-AES-GCM, because they are only supported through an external cryptographic provider. On the other hand, the SSH encryption scheme SSH-ChaCha20-Poly1305 relies entirely on a self-contained implementation.

Dropbear

Dropbear [87] also implements complete client and server-side support for the SSH version 2 protocol. Development started in October 2002 [88], and the first public release was on April 6, 2003 [88]. Amazingly, Dropbear is the product of a single developer, Matt Johnson, based in Australia (the name *dropbear* likely derives from the Australian folklore hoax animal *drop bear*, a predatory, carnivorous version of the Koala).

Dropbear is designed to be lightweight and to shine in environments where a low memory footprint is paramount. As a consequence, Dropbear doesn't include as many features as OpenSSH.

Dropbear uses LibTomCrypt [45] as its cryptographic provider, which is a required dependency.

2.4.2 SSH Binary Packet Protool

The Binary Packet Protocol (BPP) of SSH is defined in Section 6 of RFC 4253 [135]. SSH packets are constructed through a two-step process: payload encoding and cryptographic processing. These are described below. It is the BPP that defines SSH encryption schemes utilising a variety of symmetric encryption schemes (of different types) and message authentication codes.

The Binary Packet Protocol does not define any mode in which it fragments input. Other SSH service layers that makes use of the BPP as their transport layer might define such mode. For example, if the BPP is used in a data transport protocol, such mode would be defined to account for the upper limit of the size of a SSH packet. Considering this kind of input mode is out-of-scope for this thesis, see Section 3.10.4 for work that deals with this aspect in depth. In Chapter 3 we define fragmentation in the sense of ciphertext fragmentation.

Encoding

Encoding proceeds as follows. Firstly, if compression is enabled, then the payload (and only the payload) is compressed. Secondly, a length field and a padding length field are prepended to the payload and random padding appended. The length

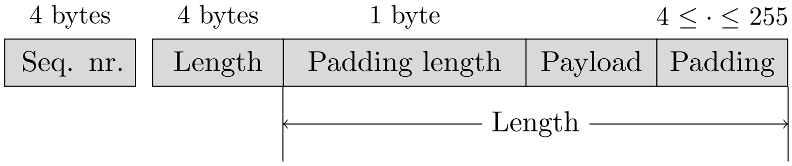

Figure 2.12: SSH BPP packet layout.

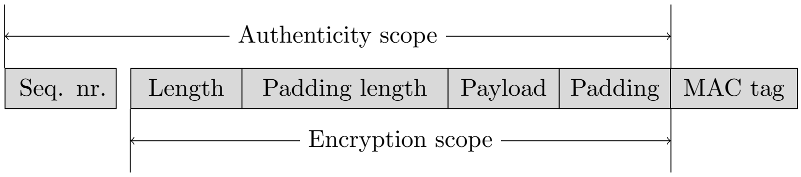

Figure 2.13: Scope of encryption and authentication for SSH BPP packets.

field has 4 bytes and encodes the combined length in bytes of the padding length field, payload and padding. The padding length field has 1 byte and encodes the length of the padding counted in bytes. The SSH standard mandates that an implementation must be able to support an uncompressed payload of at least 32,768 bytes and support a total packet length — packet length field, padding length field, uncompressed payload, padding and MAC — of at least 35,000 bytes. Padding must be between 4 and 255 bytes long and must align the packet length to a multiple of the block size of the underlying block cipher or 8, whichever is larger; stream ciphers are instantiated with a block size of 8. A 4-byte sequence number is initially set to 0 when a connection is established and is incremented by 1 for each packet sent. The sequence number is not sent over the wire, but maintained separately and included in cryptographic computations on packets. The packet layout after payload encoding is shown in Figure 2.12.

Cryptographic Processing

SSH provides confidentiality and authenticity through symmetric encryption schemes and MAC algorithms. RFC 4253 mandates that when encryption is applied, the length field, padding length field, payload and padding must be encrypted. In addition, RFC 4253 specifies that the MAC tag must be computed over the concatenation of the sequence number and the unencrypted packet, enforcing an Encrypt-and-MAC paradigm, cf. Figure 2.13.

Naming Scheme for SSH Encryption Schemes

When we refer to an SSH encryption scheme, using a symmetric encryption scheme X, and possibly, a message authentication code Y, we denote this SSH-specific scheme by SSH-X-Y. If a message authentication code is not used, we will leave out the last component of the name. The distinction between SSH encryption schemes and the underlying algorithms is important to take into account when performing an analysis of the security assurances of SSH encryption schemes, cf. Chapter 6. We emphasise the distinction by including SSH directly in the name of the scheme.

Standardised SSH Encryption Schemes

The SSH specifications define a variety of SSH encryption schemes that can be instantiated using a diverse set of symmetric encryption schemes and MAC algorithms. RFC 4253 [135] defines 7 symmetric encryption schemes (including IV and nonce-based) built from 7 block ciphers running in CBC-mode: 3DES, Blowfish, Twofish, AES, Serpent, IDEA and Cast. In CBC-mode, the IV is, for each encryption, set to be the final ciphertext block from the previous encryption; the first IV is chosen uniformly at random. As pointed out in [120, 52] for CBC-mode in general and in [20, 21] for SSH specifically, this potentially makes SSH vulnerable to chosen-plaintext attacks. However, these do not (generally) seem to be realisable in practice due to details of the SSH packet encoding.[3] Each block cipher may support several different key lengths, e.g. the AES options include support for key lengths of 128 bits and 256 bits, denoted by AES128 and AES256, respectively. Of the symmetric encryption schemes defined in RFC 4253, the scheme built from 3DES is required while the scheme built from AES128 is recommended.

[3] The chosen-plaintext attack works, at a high level, as follows. Assume you encrypt a plaintext $P = P_1 \parallel P_2 \parallel P_3$ using CBC-mode and the resulting ciphertext is $C = C_1 \parallel C_2 \parallel C_3$. Also assume that C' is the last block of the previous ciphertext (this is the "IV" used in the encryption of P). Let G be a guess of the first block P_1 of the plaintext P. Also assume that an adversary can force the encryption of $G \oplus C' \oplus C_3$ immediately after seeing C and let C_G be the resulting ciphertext (block). Since the underlying block cipher is deterministic, if $C_G = C_1$ the adversary knows that the guess G was correct. Notice it is crucial that the adversary can force the encryption of exactly $G \oplus C' \oplus C_3$. However, in SSH the plaintext being fed into the CBC encryption algorithm is encoded. Therefore, the first plaintext block fed into the CBC-mode encryption algorithm, after encoding, is actually: length $\parallel$ padding length $\parallel$ $(G \oplus C' \oplus C_3)[0:87]$, where the first two fields have length 40 bits. Note that if the adversary does not know the exact length of the plaintext P, it would not be possible to accurately determine the length field or the padding length field used in the encryption of P. [20, Section 3] describes a method that increases the likelihood of this attack succeeding, see the reference for details.

In addition to symmetric encryption schemes using block ciphers in CBC-mode, a scheme using the RC4 stream cipher is defined (for historical reasons this cipher is denoted by arcfour). RFC 4344 [13] defines CTR-mode options for all the symmetric encryption schemes mentioned in RFC 4253. Additionally, RFC 5647 [85] defines an SSH encryption scheme using AES-GCM. In this scheme, the padding length, payload and padding fields are encrypted and integrity protected. However, the length field is not encrypted but instead included as Additional Data. Formally, then, SSH-AES-GCM deviates from the requirement of RFC 4253 to encrypt the length field. This affects the security properties the SSH scheme can achieve in the ciphertext fragmentation model, cf. Chapter 3 and Chapter 6.

RFC 4253 specifies as MAC algorithms HMAC-SHA1, HMAC-SHA1-96 (output truncated to 96 bits), HMAC-MD5 and HMAC-MD5-96. Of these, support for HMAC-SHA1 is required and support for HMAC-SHA1-96 is recommended. The later RFC 6668 [34] defines HMAC-SHA2-256 and HMAC-SHA2-512, with the former being recommended and the latter optional. A draft RFC [109] specifies UMAC-32, -64, -94 and -128 for use in SSH. Here, the sequence number is passed as a nonce to UMAC.

The symmetric encryptions schemes and MAC algorithms mentioned are composed, in various (some excluded) pairs to form SSH encryption schemes. For example, no SSH RFC defines the SSH encryption scheme SSH-AES128-CBC-UMAC-32. All SSH encryption schemes are defined by static strings, which are also passed between client and server during an SSH session establishment to negotiate the specific SSH encryption scheme. We will elaborate on this further when discussing the development of new SSH encryption schemes in Section 7.8.1.

SSH Encryption Schemes in OpenSSH

OpenSSH supports additional SSH encryption scheme options beyond those specified in the general SSH RFCs. As we shall see in Chapter 4, these are quite widely adopted and therefore warrant analysis. We present such an analysis in Chapter 6. OpenSSH has also deprecated, and in some cases, completely removed support for certain SSH encryption schemes.

Since OpenSSH version 6.2, it has been possible to run supported algorithms in an Encrypt-then-MAC mode with the encryption and MAC processing being provided by any of the supported algorithms. Usage is signalled by negotiating an “etm”-MAC during key exchange (if AES-GCM or ChaCha20-Poly1305 (see below) is negotiated as symmetric encryption scheme, the special behaviour triggered when negotiating an etm-MAC is disabled). We refer to this scheme as SSH-Generic-EtM. It is instantiated by a symmetric encryption scheme and MAC. The cryptographic processing in the SSH-Generic-EtM scheme is similar to that of the SSH-AES-GCM scheme, with the length field not being encrypted but included in the MAC scope; for details, see Section 6.4.

The RFC draft [111] defines the SSH encryption scheme SSH-ChaCha20-Poly1305. This scheme combines ChaCha20, a high-speed stream cipher [28], and Poly1305MAC, a high-speed one-time message authentication code based on a design from [26]. Here, the length field is encrypted using a separate instance of ChaCha20, but the construction otherwise follows RFC 4253. For details, see Section 6.3. This option has been supported in OpenSSH since version 6.5 [69].

There has been a steady trend towards the elimination of SSH encryption schemes from OpenSSH that use weak algorithms. OpenSSH 6.7 disabled CBC-mode schemes and the RC4 scheme by default, while OpenSSH 6.9 promoted SSH-ChaCha20-Poly1305 to be the default SSH encryption scheme. OpenSSH 7.2 disabled SSH encryption schemes using Blowfish, Cast, RC4, MD5 and Truncated-HMAC. OpenSSH 7.4 disabled SSH-3DES-CBC from the client’s default proposal. OpenSSH 7.6 removed support for SSH encryption schemes using RC4 and all schemes involving Blowfish or Cast. In addition, this version removed support for HMAC-RIPEMD160 (and deprecated RSA keys of size less than 1024 bits).

OpenSSH has also seen numerous interesting additions of newer cryptographic algorithms, apart from the ones mentioned earlier. OpenSSH 4.7 [67] added support for schemes using the MAC algorithm UMAC-64 [98] and OpenSSH 6.2 [68] extended support to also include UMAC-128. Release 6.5 [69] added supported for X25119 [27] and ED25519 [30] signatures. OpenSSH release 7.7 [72] added experimental support for the post-quantum signature scheme XMSS [41] based on RFC 8391 [84]. In the 8.0 release [73], the OpenSSH developers implemented experimental support for hybrid key exchanges using the key exchange methods Streamlined-NTRU-Prime [29] and X25119. In addition, this release made generating RSA keys with a key size of 3072 bits the default.

SSH Encryption Schemes in Dropbear

Dropbear has a much less involved history of cryptographic development. Only SSH encryption schemes that use either CTR-mode or CBC-mode symmetric encryption schemes are supported, with CTR support being added in 2008 [88]. All SSH encryption schemes use the standard Encrypt-and-MAC paradigm. For CTR-mode SSH encryption schemes, block ciphers AES, 3DES and Twofish are supported. For CBC-mode SSH encryption schemes, block ciphers AES, 3DES, Twofish and Blowfish are supported. For both modes, the Twofish schemes are disabled by default.

Dropbear supports all the standardised HMAC MAC algorithms mentioned above, except HMAC-MD5-96.

The newest version of Dropbear is *2019.78* [88] (at the time of writing). It prefers the SSH encryption scheme SSH-AES128-CTR-HMAC-SHA1-96.

Chapter 3

Ciphertext Fragmentation

In this chapter, we present the theory of symmetric encryption schemes supporting ciphertext fragmentation. We formalise the setting and define several security notions, in turn, repairing a deficient notion from earlier work.

There are several security models in the literature that relate to the ciphertext fragmentation model. We review some of the related models at the end of this chapter.

3.1 On the Choice of Security Model

In this thesis, we are concerned with the analysis of the secure transport layer in SSH, defined by the various supported SSH encryption schemes. Traditionally, secure transport layers have been built using standard symmetric encryption schemes. While the former can be built from the latter, such construction is certainly non-trivial, as the latter aspires for significantly more complex functionality. Additionally, security properties of symmetric encryption schemes are modelled using atomic security models. But they do not accurately capture the security properties that SSH, and secure transport layers in general, aspire to. We elaborate on the discrepancy in the sequel.

In Section 2.3, we saw several security notions capturing confidentiality and integrity security properties of different types of symmetric encryption schemes. Common for all notions is the requirement that when decrypting a ciphertext, the entire ciphertext, and only the ciphertext, is supplied as a parameter to the decryption algorithm. We call this delivery method *atomic*, and we call security models that assume it *atomic security models*. When a security property is proved to hold in an atomic security model, an application must ensure that the decryption process respects atomic delivery. Otherwise, the proof that a security property holds might not carry over into practice. However, such an assumption is practically impossible to satisfy when applications interact with complex networks. In such networks, fragmentation can occur at any hop on the network path as a result of protocols such as TCP [116] and IP [115], fragmentation an application cannot control. Fragmentation is also often implemented by applications. This is, for example, done to avoid sending large network packets, which would otherwise rely on fragmentation mechanisms at hops on the network path to comply with physical limitations. These mechanisms are known to potentially incur unacceptable performance penalties [94]. As a result, a ciphertext is not necessarily delivered to an application in one piece but may arrive *fragmented*, and an application must cater for the fragmented delivery during decryption. We call this network/application-induced fragmentation *ciphertext fragmentation*. We next focus on SSH specifically, motivating our choice of security model.

SSH should aim to protect against replays and reordering of messages, whereas standard symmetric encryption schemes do not. Our choice of security model should capture this property naturally. Secondly, SSH has to operate over TCP/IP, which permits the delivery of ciphertexts to the receiver in an arbitrarily fragmented manner, as detailed above. Extending a standard symmetric encryption scheme to operate over TCP/IP requires intrusive changes to the decryption algorithm, to the point that even its syntax must become significantly different. It is easy to overlook this aspect and assume that such a transformation is only cosmetic and will not affect security. As a pertinent example, Bellare et al. [20] proved (in a very strong sense) security properties of a variant of SSH encryption schemes using CBC-mode. Yet, it was exactly the mechanism supporting ciphertext fragmentation that allowed the subsequent plaintext-recovery attack by Albrecht et al. [5] against both the original CBC-mode construction used in SSH and the variant proven secure by Bellare et al.

Not catering for ciphertext fragmentation is a serious shortcoming when analysing secure transport layers. We, therefore, follow the ciphertext fragmentation security model put forward in [36] (which is strongly motivated by [20, 114], see Section 3.10.1 and Section 3.10.2). The ciphertext fragmentation model uses a new symmetric encryption scheme type, named *symmetric encryption scheme supporting ciphertext fragmentation*, to capture ciphertext fragmentation functionality. This new symmetric encryption scheme type can be used to more safely build secure transport layers, see Chapter 7. We give a detailed introduction to the ciphertext fragmentation model in the rest of this chapter, as well as elaborating on several of its features.

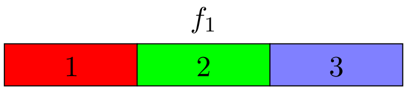

Figure 3.1: Fragment f_1 consisting of ciphertexts $c_1 = (1)$, $c_2 = (2)$ and part of ciphertext $c_3 = (34)$.

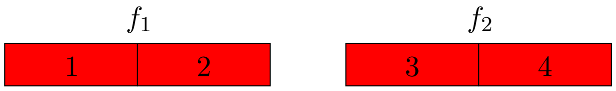

Figure 3.2: Fragments f_1 and f_2 consisting of ciphertext fragments of ciphertext $c_1 = (1234)$.

3.2 Syntax

A few difficulties arise when attempting to define a symmetric encryption scheme that natively caters for ciphertext fragmentation. Consider the situations in Figure 3.1 and Figure 3.2. In the former figure, the fragment f_1 consists of two ciphertexts c_1 and c_2, and the first component of ciphertext c_3. The decryption algorithm should be able to cut f_1 into c_1, c_2 and the partial c_3, and decrypt to messages corresponding to the decryption of c_1 and c_2. In the latter figure, a single ciphertext c_1 is divided between two fragments f_1 and f_2. The decryption algorithm must still be able to decrypt c_1 after first seeing and processing f_1 and then subsequently seeing and processing f_2. Informally, the decryption algorithm is, at a minimum, forced to support some sort of state in the sense of buffering. In [36], this minimal degree of statefulness is called *stateless beyond buffering*. We will not treat this special notion of state in this thesis.

Let $\mathsf{S}_\perp$ denote the set of errors that either the encryption or decryption algorithm can output. Note that we allow multiple errors, which reflects the fact that when a real scheme fails it might fail in different ways that can be distinguished by an adversary. This position is different than the one we take for standard symmetric encryption schemes that can only output one type of error, cf. Section 2.3. Let ¶ denote a symbol such that $\P \notin (\{0,1\} \cup \mathsf{S}_\perp)^*$. We use ¶ to indicate the end of plaintext messages. It models the need of the consuming application to correctly parse the output from the decryption algorithm into individual plaintexts.

Definition 18 (Symmetric encryption scheme supporting ciphertext fragmentation)**.** *A symmetric encryption scheme supporting ciphertext fragmentation* $\mathsf{fSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with associated key space* $\mathcal{K} \subseteq \{0,1\}^*$*, plaintext space* $\mathcal{M} \subseteq$

$\{0,1\}^*$, ciphertext space $\mathcal{C} \subseteq \{0,1\}^*$ and error set $\mathsf{S}_\bot$ ($\mathsf{S}_\bot \cap (\mathcal{M} \cup \mathcal{C}) = \emptyset$) is specified by three algorithms:

- *A randomised key generation algorithm* Gen *that outputs a key $\mathsf{k} \in \mathcal{K}$, initial encryption state σ and initial decryption state ϱ, respectively. We write $(\mathsf{k}, \sigma, \varrho) \leftarrow \mathsf{Gen}$.*
- *A probabilistic or deterministic encryption algorithm* Enc *that takes as input a key k, a plaintext $m \in \mathcal{M}$ and current encryption state σ, and outputs a ciphertext $c \in \mathcal{C} \cup \mathsf{S}_\bot$ and the updated encryption state σ'. We write $(c, \sigma') \leftarrow \mathsf{Enc_k}(m, \sigma)$.*
- *A deterministic decryption algorithm* Dec *that takes as input a key k, a ciphertext fragment $f \in \{0,1\}^*$ and current decryption state ϱ, and outputs a plaintext fragment $m \in (\{0,1,\P\} \cup \mathsf{S}_\bot)^*$ and the updated decryption state ϱ'. We write $(m, \varrho') \leftarrow \mathsf{Dec_k}(f, \varrho)$.*

Note, neither the encryption nor decryption algorithm maintain any internal state. Their entire state must be contained in their state parameter.

As for standard symmetric encryption schemes, we define a list notation. For a list $\mathcal{L}_m = [m_1, m_2, \ldots, m_l] \in \mathcal{M}^l$, we write $(\mathcal{L}_c, \sigma') \leftarrow \mathsf{Enc_k}(\mathcal{L}_m, \sigma)$ where $\mathcal{L}_c = [c_1, c_2, \ldots, c_l]$, and $(c_1, \sigma_1) \leftarrow \mathsf{Enc_k}(m_1, \sigma)$, $(c_2, \sigma_2) \leftarrow \mathsf{Enc_k}(m_2, \sigma_1)$, …, $(c_r, \sigma_l) \leftarrow \mathsf{Enc_k}(m_l, \sigma_{l-1})$, and $\sigma' = \sigma_l$. Similar notation is defined for the decryption algorithm Dec in the natural way.

There are a few things worth pointing out after seeing Definition 18. Firstly, the scheme might be made stateless by letting the key generation algorithm output empty strings instead of initial states and allowing the encryption and decryption algorithm to ignore the state. Secondly, the decryption algorithm must be able to handle fragments that decrypt to multiple plaintexts, possibly mixed with errors, or that decrypt to nothing. The latter case can occur if a fragment does not contain enough data to decrypt at all.

We assume that all symmetric encryption schemes supporting ciphertext fragmentation are length-regular. We have seen what that means for the different flavours of symmetric encryption schemes in Section 2.3. Symmetric encryption schemes supporting ciphertext fragmentation are length-regular if for all keys $\mathsf{k} \in \mathcal{K}$, that can be output by Gen, all states σ, $c \leftarrow \mathsf{Enc_k}(m, \sigma)$ and $c \notin \mathsf{S}_\bot$, then the length of

c only depends on the length of m. Hence, as for standard symmetric encryption schemes, the length of a ciphertext must be completely determined by the length of the message.

3.3 Correctness

In Section 2.3, we defined several different types of a symmetric encryption scheme. All schemes, no matter the type, are required to satisfy a correctness condition. This requirement is natural because we only want to consider symmetric encryption schemes that "make sense" and avoid degenerate cases. For the same reason, we want to focus on natural schemes that satisfy the syntax of Definition 18. However, it turns out that it is not trivial to define what natural means for schemes that support ciphertext fragmentation. In fact, there are several ways to define a correctness condition, that all "make sense". By our choice of condition it becomes clear that some sort of buffering in the decryption algorithm is necessary. First, we provide a few words of motivation, then proceed to present the correctness condition.

If ciphertexts correspond exactly to fragments, the situation is equivalent to the atomic setting. The least complicated diversion from this is when a sequence of plaintexts are encrypted, and the corresponding ciphertexts is contained in exactly one fragment, see Figure 3.3. We would then expect to see all ciphertexts decrypted when the decryption algorithm is given the fragment as input. This can be generalised to having the sequence of ciphertexts split between several fragments but always having a ciphertext boundary coincide with a fragment boundary, see Figure 3.4. The expectation is then to see all ciphertexts contained in each fragment, decrypted.

The situation becomes more intricate if ciphertext boundaries do not coincide with fragment boundaries. For example, consider the situation in Figure 3.5 where two ciphertexts $c_1 = (12)$ and $c_2 = (3)$ are split between two fragments $f_1 = (1)$ and $f_2 = (23)$. We require that after receiving the second fragment, the decryption algorithm decrypt both ciphertexts. That is, the decryption algorithm must decrypt both ciphertexts to their correct plaintexts and return the plaintexts separated by the plaintext delimiter ¶. As a last case, consider a situation where some extra data is appended to a fragment. We then require that the decryption algorithm can still produce a correct decryption of the first ciphertext. This is depicted in Figure 3.6, where a ciphertext $c = (123)$ is split between two fragments $f_1 = (1)$ and $f_2 = (234')$

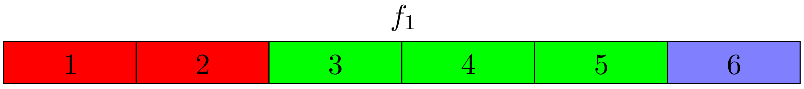

Figure 3.3: Fragment f_1 consisting of ciphertexts $c_1 = (12)$, $c_2 = (345)$ and $c_3 = (6)$.

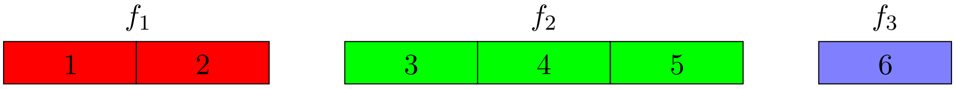

Figure 3.4: Fragments f_1, f_2 and f_3 consisting of ciphertexts $c_1 = (12)$, $c_2 = (345)$ and $c_3 = (6)$ (respectively), where ciphertext boundaries coincide with fragment boundaries.

but where some extra ciphertext has been appended to the second fragment. The ciphertext fragment might be part of a second valid ciphertext or might be invalid. In either case, it is required that the decryption algorithm decrypts the first ciphertext correctly.

For the formal definition, we define a map $\P\colon (\{0,1\}^* \cup \mathsf{S}_\perp)^* \to (\{0,1\}^* \cup \mathsf{S}_\perp \cup \{\P\})^*$ by $\P(m_1, m_2, \ldots, m_r) = m_1 \parallel \P \parallel m_2 \parallel \P \parallel \cdots \parallel m_r \parallel \P$. We trust that overloading the notation for $\P$ is not a problem. Below we give the correctness definition of a symmetric encryption scheme supporting ciphertext fragmentation. It is required throughout this thesis that any symmetric encryption scheme that supports ciphertext fragmentation satisfy this definition.

Definition 19 (Correctness)**.**
A symmetric encryption scheme supporting ciphertext fragmentation $\mathsf{fSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is correct: if for all tuples* $(\mathsf{k}, \sigma, \varrho)$ *that can be output from* Gen*, for all* $\mathcal{L}_m \in [\mathcal{M}]^*$ *and for all* $\mathcal{L}_f \in [\{0,1\}^*]^*$*, it holds true with probability one:*

$$\textit{If } (\mathcal{L}_c, \sigma') \leftarrow \mathsf{Enc}_\mathsf{k}(\mathcal{L}_m, \sigma) \textit{ and } ||(\mathcal{L}_c) \preceq ||(\mathcal{L}_f) \textit{ and } (\mathcal{L}_{m_f}, \varrho_f) \leftarrow \mathsf{Dec}_\mathsf{k}(\mathcal{L}_f, \varrho),$$
$$\textit{then } \P(\mathcal{L}_m) \preceq ||(\mathcal{L}_{m_f}).$$

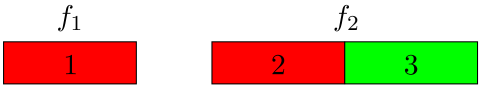

Figure 3.5: Fragments f_1 and f_2 consisting of ciphertexts $c_1 = (12)$ and $c_2 = (3)$.

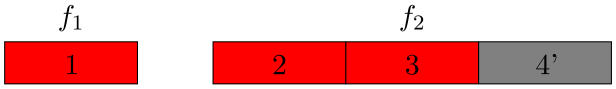

Figure 3.6: Fragments f_1 and f_2 consisting of ciphertext $c_1 = (123)$ and extra data $(4')$.

3.3.1 Alternative Correctness Definitions

The situation arising in Figure 3.6 illustrates one choice that has to be made when defining correctness: how should the extra data be handled? Definition 19 requires that any excess ciphertext is carried over to the next execution of the decryption algorithm to allow correct decryption when a ciphertext is split between two or more fragments (making buffering necessary). This is one particular interpretation of "correct". Below, we consider two alternative interpretations.

Flush Ignore any surplus ciphertext, and return the decrypted plaintext(s). This interpretation has the benefit of removing the requirement for any buffering between two distinct decryption calls.

Fault Return an error if a fragment contains partial ciphertext, deeming the fragment invalid.

The two interpretations above both represent a well-defined correctness condition. However, we opted for the least restrictive correctness definition, and allow fragments to extend over multiple ciphertexts. This is also the choice made in [36].

3.4 Confidentiality

In this section, confidentiality of messages in the presence of fragmentation is defined. It is obvious that any conventional confidentiality notion for symmetric encryption schemes that do not include the decryption algorithm, such as IND-CPA, can be used for the fragmentation case as well (except one has to use a stateful version of the security notion, see Section 3.10.1). This is no longer the case for IND-CCA style notions where we have to take into account the special behaviour of the decryption algorithm. Such a notion was first formalised in [36], denoted by IND-sfCFA. Unfortunately, the notion is unsatisfiable as presented because of a generic attack.

```
alg. $\mathsf{INI}$
  $\mathsf{sync} = \mathsf{true}$
  $i_e = 0$
  $\mathcal{L}_{\mathsf{C}} = []$
  $\mathcal{L}_{\mathsf{M}} = []$
  $\mathsf{S_D} = \epsilon$
  $\mathsf{S_F} = \epsilon$
  $b \leftarrow\!\$ \{0,1\}$
  $(\mathsf{k}, \sigma, \varrho) \leftarrow \mathsf{Gen}$
  return
```

```
alg. $\mathsf{LR}(b, M_0, M_1)$
  if $|M_0| \neq |M_1|$
    return $\perp$
  $(C, \sigma') \leftarrow \mathsf{Enc_k}(M_b, \sigma)$
  $\mathcal{L}_{\mathsf{C}}.\mathsf{append}(C)$
  $\mathcal{L}_{\mathsf{M}}.\mathsf{append}(M_b \parallel \P)$
  $i_e = i_e + 1$
  return $C$
```

```
alg. $\mathsf{DEC}(F)$
  $(M, \varrho') \leftarrow \mathsf{Dec_k}(F, \varrho)$
  $\mathsf{S_F} = \mathsf{S_F} \parallel F$
  $\mathsf{S_D} = \mathsf{S_D} \parallel M$
  if $\mathsf{sync} = \mathsf{true}$
    $\mathsf{U} = \{l \mid ||(\mathcal{L}_{\mathsf{C}}[0:l]) \npreceq \mathsf{S_F}\} \cup \{i_e\}$
    $j_d = \min(\mathsf{U})$
    if $\mathsf{S_F} \preceq ||(\mathcal{L}_{\mathsf{C}}[0:j_d])$
      $M = \epsilon$
    else
      $M = \mathsf{S_D} \,\%\, ||(\mathcal{L}_{\mathsf{M}}[0:j_d - 1])$
      if $||(\mathcal{L}_{\mathsf{C}}[0:j_d]) \preceq \mathsf{S_F}$
        $M = \mathsf{S_D} \,\%\, ||(\mathcal{L}_{\mathsf{M}}[0:j_d])$
      if $M \neq \epsilon$
        $\mathsf{sync} = \mathsf{false}$
  return $M$
```

Figure 3.7: Algorithms for defining IND-sfCFA advantage.

We will treat this issue in depth in Section 3.8. We propose a definition that fixes the deficiency, and it is this definition we will use throughout.

Definition 20 (IND-sfCFA).
Let $\mathsf{fSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme supporting ciphertext fragmentation. Let* LR *and* DEC*, all being initialised by* INI*, be the algorithms defined in Figure 3.7. For any adversary* $\mathcal{A}$*, we define its* IND-sfCFA *advantage as:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}sfcfa}}_{\mathsf{fSE}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{INI} \,:\, \mathcal{A}^{\mathsf{LR}(b,\cdot,\cdot),\mathsf{DEC}(\cdot)} = b\right] - 1.$$

The scheme fSE *is* (η, R)-IND-sfCFA *secure if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* IND-sfCFA *advantage is bounded by* η*.*

The encryption oracle LR encrypts a plaintext, depending on a bit b and appends the plaintext and resulting ciphertext to two separate lists. This information is used in the decryption oracle to decide when to start releasing plaintext (i.e. when to go out-of-sync) and to decide what to release. It then returns the resulting ciphertext, which might be an error, and increments the encryption invocation counter by one.

The decryption oracle DEC is a complex oracle responsible for detecting attack attempts and ruling out trivial wins. An adversary is allowed to query any ciphertext fragment to the decryption oracle, including ciphertext output by the encryption oracle. To avoid trivial wins the decryption oracle suppresses output when ciphertext fragment queries do not deviate (i.e. are in-sync) from the sequence of ciphertexts output by the encryption oracle. When the decryption oracle goes out-of-sync, plaintext produced from in-sync ciphertexts will be discarded from the output, again to avoid trivial wins. Detecting a deviation correctly and discarding plaintexts produced from in-sync ciphertexts, is made difficult by the complex relationship between adversarially chosen-ciphertext fragments and the original ciphertext output by the encryption oracle. However, it is crucial that the decryption oracle allows in-sync queries in order to capture stateful security (e.g. to allow an adversary to exploit weaknesses related to the state of the decryption algorithm by first querying in-sync ciphertexts). If a deviation is detected, plaintext is returned depending on the reason for the deviation. Essentially, if the deviation occurred because of a bit difference in the two ciphertext sequences (output from encryption oracle and queries to the decryption oracle), then $\mathsf{S}_\mathsf{D}\,\%\,||(\mathcal{L}_\mathsf{M}[0:j_d-1])$ will be returned. But, if the deviation is a result of the adversary submitting more ciphertext than has been produced by the encryption oracle, then $\mathsf{S}_\mathsf{D}\,\%\,||(\mathcal{L}_\mathsf{M}[0:j_d])$ will be returned. Boldyreva et al. [36] missed the last case making a trivial win possible, see Section 3.8 for details.

Traditional confidentiality security notions that use indistinguishability, such as IND-CPA, have results showing equivalence to semantic security (first defined in [78]). Semantic security precisely describes the security property, but with the downside of being difficult to work with. The indistinguishably approach is much easier to work with, but lacks the clear intuition the semantic definition gives. In the fragmentation case, we only have the indistinguishably definition and lack a more intuitive definition (possibly together with an equivalence proof) that can show we have found the "correct" definition. Furthermore, traditional notions, by virtue of being traditional, have stood the test of time and can be considered sound security notions. No security notion in the ciphertext fragmentation model has this luxury; they have all yet to stand the test of time.

3.5 Authenticity

In the original work [36], no definition of authenticity is given. We give a definition that follows the approach of the INT-CTXT notion (Definition 7).

Definition 21 (INT-sfCTF).

```
alg. INI
  $\mathsf{sync} = \mathsf{true}$
  $i_e = 0$
  $\mathcal{L}_{\mathsf{C}} = []$
  $\mathcal{L}_{\mathsf{M}} = []$
  $\mathsf{S_D} = \epsilon$
  $\mathsf{S_F} = \epsilon$
  $(\mathsf{k}, \sigma, \varrho) \leftarrow \mathsf{Gen}$
  return
```

```
alg. ENC($M$)
  $(C, \sigma') \leftarrow \mathsf{Enc_k}(M, \sigma)$
  $\mathcal{L}_{\mathsf{C}}.\mathsf{append}(C)$
  $\mathcal{L}_{\mathsf{M}}.\mathsf{append}(M \parallel ¶)$
  $i_e = i_e + 1$
  return $C$
```

```
alg. DEC($F$)
  $(M, \varrho') \leftarrow \mathsf{Dec_k}(F, \varrho)$
  $\mathsf{S_F} = \mathsf{S_F} \parallel F$
  $\mathsf{S_D} = \mathsf{S_D} \parallel M$
  if $\mathsf{sync} = \mathsf{true}$
    $\mathsf{U} = \{l \mid ||(\mathcal{L}_{\mathsf{C}}[0:l]) \npreceq \mathsf{S_F}\} \cup \{i_e\}$
    $j_d = \min(\mathsf{U})$
    if $\mathsf{S_F} \preceq ||(\mathcal{L}_{\mathsf{C}}[0:j_d])$
      $M = \epsilon$
    else
      $M = \mathsf{S_D} \,\%\, ||(\mathcal{L}_{\mathsf{M}}[0:j_d - 1])$
      if $||(\mathcal{L}_{\mathsf{C}}[0:j_d]) \preceq \mathsf{S_F}$
        $M = \mathsf{S_D} \,\%\, ||(\mathcal{L}_{\mathsf{M}}[0:j_d])$
      if $M \neq \epsilon$
        $\mathsf{sync} = \mathsf{false}$
  return $M$
```

Figure 3.8: Algorithms for defining INT-sfCTF advantage.

Let $\mathsf{fSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme supporting ciphertext fragmentation. Let* ENC *and* DEC*, both being initialised by* INI*, be the algorithms defined in Figure 3.8. Let* FORGE *be the event that* DEC *returns an element from* $\{0, 1, ¶\}^+$*. For any adversary* $\mathcal{A}$*, we define its* INT-sfCTF *advantage as:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}ctf}}_{\mathsf{fSE}}(\mathcal{A}) = \Pr\Big[\mathsf{INI},\ \mathcal{A}^{\mathsf{ENC}(\cdot),\mathsf{DEC}(\cdot)} : \mathsf{FORGE}\Big].$$

The scheme fSE *is* (η, R)-INT-sfCTF *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* INT-sfCTF *advantage is bounded by* η*.*

The security game played is essentially the same security game played in the IND-sfCFA game. But instead of guessing a bit, we measure the ability of an adversary to produce a ciphertext that makes DEC output an element from $\{0, 1, ¶\}^+$. Note that the adversary can choose any ciphertext, even ciphertexts that have been output by the encryption oracle. Hence, to meet the authenticity notion INT-sfCTF, the decryption algorithm is normally required to include the state in the authenticity protection, in some way. For example, the state could be included in a MAC computation.

3.6 Boundary Hiding

It is standard to assume that a symmetric encryption scheme is allowed to leak the length of an underlying plaintext, which is also the convention followed in this thesis.

However, leaking the length of a plaintext can be potentially dangerous [54, 55, 130]. The SSH Binary Packet Protocol (cf. Section 2.4.2) attempts to hide plaintext lengths by encrypting metadata such as the packet length field. Other protocols such as TLS have an explicit length field and therefore do not achieve any hiding of plaintext lengths. Boldyreva et al. [36] formalised boundary hiding security notions for passive and active adversaries. These notions provide a form of length hiding for sequences of plaintexts, intended to capture formally what SSH attempts to achieve (see Section 6.6 for a bit more detail on this). They are replicated below.

Definition 22 (BH-CPA).
Let $\mathsf{fSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme supporting ciphertext fragmentation. Let* $\mathsf{LR\text{-}BH}$*, initialised by* INI*, be the algorithms defined in Figure 3.9. For any adversary* $\mathcal{A}$*, we define its* $\mathsf{BH\text{-}CPA}$ *advantage as:*

$$\mathsf{Adv}^{\mathsf{bh\text{-}cpa}}_{\mathsf{fSE}}(\mathcal{A}) = 2 \cdot \Pr\Big[\mathsf{INI} \; : \; \mathcal{A}^{\mathsf{LR\text{-}BH}(b,\cdot,\cdot)} = b\Big] - 1.$$

The scheme fSE *is* (η, R)-$\mathsf{BH\text{-}CPA}$ *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* $\mathsf{BH\text{-}CPA}$ *advantage is bounded by* η*.*

Definition 23 (BH-sfCFA).
Let $\mathsf{fSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme supporting ciphertext fragmentation. Let* $\mathsf{LR\text{-}BH}$ *and* DEC*, both being initialised by* INI*, be the algorithms defined in Figure 3.9. For any adversary* $\mathcal{A}$*, we define its* $\mathsf{BH\text{-}sfCFA}$ *advantage as:*

$$\mathsf{Adv}^{\mathsf{bh\text{-}sfcfa}}_{\mathsf{fSE}}(\mathcal{A}) = 2 \cdot \Pr\Big[\mathsf{INI} \; : \; \mathcal{A}^{\mathsf{LR\text{-}BH}(b,\cdot,\cdot),\mathsf{DEC}(\cdot)} = b\Big] - 1.$$

The scheme fSE *is* (η, R)-$\mathsf{BH\text{-}sfCFA}$ *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* $\mathsf{BH\text{-}sfCFA}$ *advantage is bounded by* η*.*

The passive boundary hiding game BH-CPA captures the case where an adversary tries to gather information from ciphertext lengths. This property is quantified by an indistinguishability game, where an adversary can ask a left-or-right oracle to encrypt one of two sequences of plaintexts, as long as the total length of the resulting ciphertexts is the same for both sequences. To win, the adversary must distinguish which sequence is being encrypted. Note, that we do not require that the two plaintext sequences contain the same number of plaintexts, nor that the plaintexts have the same lengths. Therefore, intuitively, a scheme that does not hide plaintext lengths (e.g. appending a length field to the ciphertext) cannot satisfy the

```
alg. $\mathsf{INI}$

$\mathsf{sync} = \mathsf{true}$
$i_e = 0$
$\mathcal{L}_{\mathsf{C}} = []$
$\mathcal{L}_{\mathsf{M}} = []$
$\mathsf{S}_{\mathsf{D}} = \epsilon$
$\mathsf{S}_{\mathsf{F}} = \epsilon$
$b \leftarrow\$ \{0,1\}$
$(\mathsf{k}, \sigma, \varrho) \leftarrow \mathsf{Gen}$
return
```

```
alg. $\mathsf{LR\text{-}BH}(b, \mathcal{L}_{M_0}, \mathcal{L}_{M_1})$

$(\mathcal{L}_{C_0}, \sigma'_0) \leftarrow \mathsf{Enc}_{\mathsf{k}}(\mathcal{L}_{M_0}, \sigma)$
$(\mathcal{L}_{C_1}, \sigma'_1) \leftarrow \mathsf{Enc}_{\mathsf{k}}(\mathcal{L}_{M_1}, \sigma)$
$C_0 = ||(\mathcal{L}_{C_0})$
$C_1 = ||(\mathcal{L}_{C_1})$
if $|C_0| \neq |C_1|$
  return $\perp$
$\sigma' = \sigma'_b$
for $k = 0 \ldots |\mathcal{L}_{C_b}| - 1$
  $i_e = i_e + 1$
  $\mathcal{L}_{\mathsf{C}}.\mathsf{append}(\mathcal{L}_{C_b}[k])$
  $\mathcal{L}_{\mathsf{M}}.\mathsf{append}(\mathcal{L}_{M_b}[k] \parallel \P)$
return $\mathcal{L}_{C_b}$
```

```
alg. $\mathsf{DEC}(F)$

$(M, \varrho') \leftarrow \mathsf{Dec}_{\mathsf{k}}(F, \varrho)$
$\mathsf{S}_{\mathsf{F}} = \mathsf{S}_{\mathsf{F}} \parallel F$
$\mathsf{S}_{\mathsf{D}} = \mathsf{S}_{\mathsf{D}} \parallel M$
if $\mathsf{sync} = \mathsf{true}$
  $\mathsf{U} = \{l \mid ||(\mathcal{L}_{\mathsf{C}}[0:l]) \not\preceq \mathsf{S}_{\mathsf{F}}\} \cup \{i_e\}$
  $j_d = \min(\mathsf{U})$
  if $\mathsf{S}_{\mathsf{F}} \preceq ||(\mathcal{L}_{\mathsf{C}}[0:j_d])$
    $M = \epsilon$
  else
    $M = \mathsf{S}_{\mathsf{D}} \,\%\, ||(\mathcal{L}_{\mathsf{M}}[0:j_d - 1])$
    if $||(\mathcal{L}_{\mathsf{C}}[0:j_d]) \preceq \mathsf{S}_{\mathsf{F}}$
      $M = \mathsf{S}_{\mathsf{D}} \,\%\, ||(\mathcal{L}_{\mathsf{M}}[0:j_d])$
    if $M \neq \epsilon$
      $\mathsf{sync} = \mathsf{false}$
return $M$
```

Figure 3.9: Algorithms for defining BH-CPA and BH-sfCFA advantage.

BH-CPA notion. The active boundary hiding game BH-sfCFA captures an adversary that tries to infer information by means of, for example, flipping bits in the stream of fragments. An adversary is both given access to the left-or-right oracle as in the BH-CPA security game, as well as a decryption oracle. Note that neither definition attempts to capture side-channel attacks based on, for example, the time taken to process ciphertext fragments. Rather, they are focussed on information that leaks directly to the adversary via the ciphertexts themselves, or from error messages received during decryption. Nevertheless, as we show in Chapter 6, none of the SSH encryption schemes currently supported in OpenSSH achieves BH-sfCFA security due to simple bit-flipping attacks. This is likely true for all SSH implementations, but we have not investigated this further. In Section 7.6, we discuss this topic further with respect to libInterMAC, a library implementing a symmetric encryption scheme supporting ciphertext fragmentation, that meets both BH-CPA and BH-sfCFA.

Below we give two relations that will become useful later on. The first theorem says that if a symmetric encryption scheme supporting ciphertext fragmentation is length-regular and has ciphertexts that are indistinguishable from random strings, then the scheme also (passively) hides ciphertext boundaries. For this theorem, we assume that the notion IND$-CPA defined in Definition 5 is extended to cover symmetric encryption schemes supporting ciphertext fragmentation. The second theorem says that for length-regular schemes, (active) boundary hiding implies left-or-right indistinguishability in the sense of $\mathsf{IND\text{-}sfCFA}$.

Theorem 2.
Let $\mathsf{fSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a (length-regular) symmetric encryption scheme supporting ciphertext fragmentation. Then for any* $\mathsf{BH\text{-}CPA}$ *adversary* $\mathcal{A}_{\mathsf{bhcpa}}$ *against* fSE*, there exists an* $\mathsf{IND\$\text{-}CPA}$ *adversary* $\mathcal{A}_{\mathsf{\$cpa}}$ *against* fSE *such that:*

$$\mathsf{Adv}^{\mathsf{bh\text{-}cpa}}_{\mathsf{fSE}}(\mathcal{A}_{\mathsf{bhcpa}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{ind\$\text{-}cpa}}_{\mathsf{fSE}}(\mathcal{A}_{\mathsf{\$cpa}}).$$

If $\mathcal{A}_{\mathsf{bhcpa}}$ *makes* q_e *encryption queries totalling* μ_e *number of bits, then* $\mathcal{A}_{\mathsf{\$cpa}}$ *makes at most* μ_e *encryption queries totalling at most* μ_e *number of bits.*

Proof. Since fSE is length-regular, there exists a function f such that $f(m)$ is the length of the ciphertext under the encryption of m. For a list $\mathcal{L}_m$, we let $f(\mathcal{L}_m)$ denote the sum of $f(m')$ over strings m' contained in $\mathcal{L}_m$. For any adversary $\mathcal{A}_{\mathsf{bhcpa}}$, we construct $\mathcal{A}_{\mathsf{\$cpa}}$ as follows: Adversary $\mathcal{A}_{\mathsf{\$cpa}}$ picks a bit d uniformly at random and runs $\mathcal{A}_{\mathsf{bhcpa}}$. Then on an encryption query $(\mathcal{L}_{m_0}, \mathcal{L}_{m_1})$, $\mathcal{A}_{\mathsf{\$cpa}}$ simulates the special left-or-right encryption oracle to $\mathcal{A}_{\mathsf{bhcpa}}$ by first checking that $f(\mathcal{L}_{m_0}) = f(\mathcal{L}_{m_1})$ and then uses the bit d to pick $\mathcal{L}_{m_d}$, encrypting each list component-wise (by querying its $\mathsf{IND\$\text{-}CPA}$ encryption oracle), and finally, returning the concatenation of the resulting ciphertexts. If the first length check fails, $\perp$ is returned instead. If $\mathcal{A}_{\mathsf{bhcpa}}$ outputs d then $\mathcal{A}_{\mathsf{\$cpa}}$ outputs 1 and 0 otherwise. If the $\mathsf{IND\$\text{-}CPA}$ oracle is ENC, $\mathcal{A}_{\mathsf{\$cpa}}$ provides $\mathcal{A}_{\mathsf{bhcpa}}$ with a perfect simulation of the $\mathsf{BH\text{-}CPA}$ game with a random bit d. If the $\mathsf{IND\$\text{-}CPA}$ oracle is $\$$, the replies to the oracle queries by $\mathcal{A}_{\mathsf{bhcpa}}$ will be independent of d. In turn, the output from $\mathcal{A}_{\mathsf{bhcpa}}$ is equal to d with probability 1/2. Hence, $\mathcal{A}_{\mathsf{\$cpa}}$ outputs 1 with probability 1/2. Thus

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{ind\$\text{-}cpa}}_{\mathsf{fSE}}(\mathcal{A}_{\mathsf{\$cpa}}) &= \Pr\Big[\mathsf{INI} : \mathcal{A}^{\mathsf{ENC}(\cdot)}_{\mathsf{\$cpa}} = 1\Big] - \Pr\Big[\mathsf{INI} : \mathcal{A}^{\mathbf{\$}(\cdot)}_{\mathsf{\$cpa}} = 1\Big] \\
&= \Pr\Big[\mathsf{INI} : \mathcal{A}^{\mathsf{LR\text{-}BH}(d,\cdot,\cdot)}_{\mathsf{bhcpa}} = d\Big] - \Pr\Big[\mathsf{INI} : \mathcal{A}^{\mathbf{\$}(\cdot)}_{\mathsf{\$cpa}} = 1\Big] \\
&= \Pr\Big[\mathsf{INI} : \mathcal{A}^{\mathsf{LR\text{-}BH}(d,\cdot,\cdot)}_{\mathsf{bhcpa}} = d\Big] - \frac{1}{2} \\
&= \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{bh\text{-}cpa}}_{\mathsf{fSE}}(\mathcal{A}_{\mathsf{bhcpa}}).
\end{aligned}$$

Note that each encryption query by $\mathcal{A}_{\mathsf{bhcpa}}$ can consist of one list that is empty and one list that consist entirely of 1-bit plaintexts. Since $\mathcal{A}_{\mathsf{\$cpa}}$ is forced to encrypt a list component-wise, the resource consumption follows. □

The theorem above first appeared in [36], but with the difference that the encryption scheme fSE was not assumed to be length-regular. This is, however, required. The proof constructs a reduction from $\mathsf{BH\text{-}CPA}$ to $\mathsf{IND\$\text{-}CPA}$. The reduction must ensure that any encryption oracle queries from the boundary hiding adversary must produce ciphertexts of equal total length. Assuming length regularity ensures that the reduction can enforce the condition. Furthermore, the original theorem stated that the resource consumption of the reduction is similar to the assumed adversary. This is not true in general, as can be seen from the proof.

The implication in Theorem 2 is strict. This can be seen by applying an extra encoding of the ciphertexts that does not break the boundary hiding property. Take the ciphertext output from the encryption algorithm and re-encode it by doubling any bit. That is, apply the following maps to each bit of the ciphertext: $0 \to 00$ and $1 \to 11$. Clearly, the augmented scheme still achieves boundary hiding, but the ciphertexts are certainly not indistinguishable from a random string (of the same length).

Theorem 3.
Let $\mathsf{fSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a (length-regular) symmetric encryption scheme supporting ciphertext fragmentation. Then for any* $\mathsf{IND\text{-}sfCFA}$ *adversary* $\mathcal{A}_{\mathsf{sfcfa}}$ *against* fSE*, there exists a* $\mathsf{BH\text{-}sfCFA}$ *adversary* $\mathcal{A}_{\mathsf{bhcfa}}$ *against* fSE *such that:*

$$\mathsf{Adv}_{\mathsf{fSE}}^{\mathsf{ind\text{-}sfcfa}}(\mathcal{A}_{\mathsf{sfcfa}}) \leq \mathsf{Adv}_{\mathsf{fSE}}^{\mathsf{bh\text{-}sfcfa}}(\mathcal{A}_{\mathsf{bhcfa}}),$$

where adversary $\mathcal{A}_{\mathsf{bhcfa}}$ *consumes resources equal to* $\mathcal{A}_{\mathsf{sfcfa}}$*.*

Proof. The proof is straightforward. For any adversary $\mathcal{A}_{\mathsf{sfcfa}}$, we construct $\mathcal{A}_{\mathsf{bhcfa}}$ as follows: Adversary $\mathcal{A}_{\mathsf{bhcfa}}$ runs $\mathcal{A}_{\mathsf{sfcfa}}$. $\mathcal{A}_{\mathsf{bhcfa}}$ forwards any encryption queries from $\mathcal{A}_{\mathsf{sfcfa}}$ to its own encryption oracle, after checking that the bit-length of the two plaintexts is the same. If the length of the two plaintexts is not equal, $\mathcal{A}_{\mathsf{bhcfa}}$ returns the error $\perp$. Likewise, $\mathcal{A}_{\mathsf{bhcfa}}$ forwards any decryption queries from $\mathcal{A}_{\mathsf{sfcfa}}$ to its own decryption oracle. Note that since the length of the two plaintexts is the same and fSE is length-regular, the bit-length of the corresponding ciphertexts will be the same. $\mathcal{A}_{\mathsf{bhcfa}}$ returns the same bit as $\mathcal{A}_{\mathsf{sfcfa}}$ returns. This provides $\mathcal{A}_{\mathsf{sfcfa}}$ with a perfect simulation of $\mathsf{IND\text{-}sfCFA}$. We thus have that:

$$\mathsf{Adv}_{\mathsf{fSE}}^{\mathsf{bh\text{-}sfcfa}}(\mathcal{A}_{\mathsf{bhcfa}}) = \mathsf{Adv}_{\mathsf{fSE}}^{\mathsf{ind\text{-}sfcfa}}(\mathcal{A}_{\mathsf{sfcfa}}).$$

□

The reverse of Theorem 3 is not true: append a special marker to each ciphertext, e.g., 1^{128}. An adversary can then, with high probability, delineate ciphertext boundaries by searching for the special marker string, while the scheme remains $\mathsf{IND\text{-}sfCFA}$ secure.

3.7 Denial-of-Service

Fragmentation can aid in the successful execution of Denial-of-Service (DoS) attacks. Consider, for example, the SSH protocol and its conventional SSH encryption schemes that use encryption modes such as CBC-mode or CTR-mode. By flipping certain bits corresponding to the packet length field of an SSH packet, the receiver of the packet can be tricked into believing that the message being received is many times bigger than its actual size. A user would experience such attacks as connection hanging – effectively a DoS. A cryptographic-style definition of DoS attacks was (again) first formalised in [36] and is somewhat tailored to capture the kind of attack described above.

Definition 24 ($\mathsf{DOS\text{-}sfCFA}$)**.**
Let $\mathsf{fSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme supporting fragmentation. Let* ENC *and* $\mathsf{DEC\text{-}DOS}$*, both initialised by* INI*, be the algorithms defined in Figure 3.10. Let* DENIAL *be the event that* $\mathsf{SUCCESS} = \mathsf{true}$ *after a call to* $\mathsf{DEC\text{-}DOS}$*. For any adversary* $\mathcal{A}$*, we define its* n-$\mathsf{DOS\text{-}sfCFA}$ *advantage as:*

$$\mathsf{Adv}^{\mathsf{n\text{-}dos\text{-}sfcfa}}_{\mathsf{fSE}}(\mathcal{A}) = \Pr\Big[\mathsf{INI},\ \mathcal{A}^{\mathsf{ENC}(\cdot),\mathsf{DEC\text{-}DOS}(\cdot)}\ :\ \mathsf{DENIAL}\Big]\,.$$

The scheme fSE *is said to be* (η, R)-n-$\mathsf{DOS\text{-}sfCFA}$ *secure, if for any adversary* $\mathcal{A}$ *with resources at most* R*, its* n-$\mathsf{DOS\text{-}sfCFA}$ *advantage is bounded by* η*.*

To win the DoS game, an adversary must produce a sequence of ciphertext fragments that when concatenated is at least n bits long and deviates from the sequence of bits produced by the encryption oracle, and where the consecutive decryption of these fragments still causes the decryption algorithm to produce no output. The parameter n quantifies for how long the output from the decryption algorithm can be stalled. Hence, we are interested in achieving n-$\mathsf{DOS\text{-}sfCFA}$ for the smallest possible n. Applications often implement a maximum message size to thwart DoS attempts. Such applications can trivially satisfy this definition by simply choosing n

```
alg. INI
$\mathsf{sync} = \mathsf{true}$
$i_e = 0$
$\mathcal{L}_\mathsf{C} = []$
$\mathcal{L}_\mathsf{M} = []$
$\mathsf{S_F} = \epsilon$
$\mathsf{S_D} = \epsilon$
$(\mathsf{k}, \sigma, \varrho) \leftarrow \mathsf{Gen}$
$\mathsf{SUCCESS} = \mathsf{false}$
return
```

```
alg. ENC($M$)
$(C, \sigma') \leftarrow \mathsf{Enc_k}(M, \sigma)$
$\mathcal{L}_\mathsf{C}.\mathsf{append}(C)$
$\mathcal{L}_\mathsf{M}.\mathsf{append}(M \parallel \P)$
$i_e = i_e + 1$
return $C$
```

```
alg. DEC-DOS($F$)
$(M, \varrho') \leftarrow \mathsf{Dec_k}(F, \varrho)$
$\mathsf{S_D} = \mathsf{S_D} \parallel M$
if $\mathsf{sync} = \mathsf{true}$
  $\mathsf{S_F} = \mathsf{S_F} \parallel F$
  $\mathsf{U} = \{l \,|\, ||(\mathcal{L}_\mathsf{C}[0:l]) \not\preceq \mathsf{S_F}\} \cup \{i_e\}$
  $j_d \leftarrow \min(\mathsf{U})$
  if $\mathsf{S_F} \preceq ||(\mathcal{L}_\mathsf{C}[0:j_d])$
    $M = \epsilon$
  else
    $M = \mathsf{S_D} \,\%\, ||(\mathcal{L}_\mathsf{M}[0:j_d - 1])$
    if $||(\mathcal{L}_\mathsf{C}[0:j_d]) \preceq \mathsf{S_F}$
      $M = \mathsf{S_D} \,\%\, ||(\mathcal{L}_\mathsf{M}[0:j_d])$
    if $M \neq \epsilon$
      $\mathsf{S_F} = \epsilon$
      $\mathsf{sync} = \mathsf{false}$
else
  if $M = \epsilon$
    $\mathsf{S_F} = \mathsf{S_F} \parallel F$
  else
    $\mathsf{S_F} = \epsilon$
if $\mathsf{sync} = \mathsf{false}$ and $|\mathsf{S_F}| \geq n$
  $\mathsf{SUCCESS} = \mathsf{true}$
return $M$
```

Figure 3.10: Algorithms for defining DOS-sfCFA advantage.

to be equal to the maximum message size. In Chapter 7, we present an encryption scheme supporting ciphertext fragmentation, InterMAC, that substantially lowers the smallest possible n, often far lower than any maximum message size, significantly improving DoS security. Note that in the context of the security definition, n is counted in bits.

3.8 A Deficient Definition of Confidentiality

The original definition of IND-sfCFA, as it appeared in [36], had a flaw, that allowed a trivial attack to succeed with very high probability. We have replicated the original decryption oracle in Figure 3.11. Essentially, the flaw originated from the choice of how to handle ciphertext in the decryption algorithm that deviates from the encryption oracle output. Boldyreva et al. adopted the choice that synchronisation should be lost from the moment a single bit, that has not been honestly produced through the encryption oracle, is submitted to the decryption algorithm. Unfortunately, this

```
alg. $\mathsf{sfDec}(F)$ from [36]
$(M, \varrho') \leftarrow \mathsf{Dec_k}(F, \varrho)$
$\mathsf{S_F} = \mathsf{S_F} \,\|\, F$
$\mathsf{S_D} = \mathsf{S_D} \,\|\, M$
if $\mathsf{sync} = \mathsf{true}$
  while $||(\mathcal{L}_\mathsf{C}[0 : j_d]) \preceq \mathsf{S_F}$ and $j_d < i_e$
    $j_d = j_d + 1$
  if $\mathsf{S_F} \preceq ||(\mathcal{L}_\mathsf{C}[0 : j_d])$
    $M = \epsilon$
  else
    $\mathsf{sync} = \mathsf{false}$
    $M = \mathsf{S_D} \,\%\, ||(\mathcal{L}_\mathsf{M}[0 : j_d - 1])$
return $M$
```

Figure 3.11: The decryption oracle used in the deficient IND-sfCFA notion in [36].

allows for the following attack:

1. The adversary picks a uniformly random bit d and queries the bit to the decryption oracle sfDec.
2. The adversary picks two different plaintexts m_0 and m_1 and queries these to the encryption oracle LR and receives back the resulting ciphertext c that is an encryption of m_b.
3. If $c[0] \neq d$ (i.e. the first bit of c is not equal to the bit d), then the adversary outputs a random bit. If instead $c[0] = d$, then the adversary queries $c[1 : |c|-1]$ to the decryption oracle, receiving back a plaintext m'.
4. If $m' = m_0$ the adversary returns 0, and 1 otherwise.

After the decryption oracle query in step (1), the sync flag sync is set to false in Figure 3.11 because $j_d = i_e$ and the list $\mathcal{L}_\mathsf{C}$ is empty. On the subsequent query in step (3), the oracle sfDec, therefore, returns the output from decrypting c, given the guess d was correct. Step (4) will produce the correct guess of the bit b because of correctness. Since each case in step 3 occurs with probability 1/2, the success probability of the adversary is $1/2 + 1/4 = 3/4$. Hence, the adversary's advantage is 1/2.

The attack succeeds by cleverly abusing the internal buffering of the decryption algorithm Dec and the decryption oracle's handling of the dishonestly produced ciphertext fragment. To fix this flaw, we choose to adopt a stricter interpretation of when the decryption oracle should go out-of-sync. In essence, we believe that the decryption gives too much credit to an adversary that can guess a small prefix of the “next” ciphertext fragment. We, therefore, require that out-of-sync is only achieved if the adversary can force output from the decryption oracle, different from the empty string after honest plaintext has been removed from the output buffer $\mathsf{S_D}$. This removes the ability of the adversary to directly take advantage of the inherent buffering between calls, without showing that it can achieve a genuine advantage. The honest plaintext removed from the output buffer depends on how the deviation occurs and will be either $||(\mathcal{L}_\mathsf{M}[0\ldots j_d - 1])$ or $||(\mathcal{L}_\mathsf{M}[0\ldots j_d])$. The latter case occurs if $||(\mathcal{L}_\mathsf{C})$ is a (strict) prefix of $\mathsf{S_F}$ and the former occurs if a bit is different in at least one index in both string $||(\mathcal{L}_\mathsf{C})$ and string $\mathsf{S_F}$.

The new definition is more intricate than the original definition. Note, for example, that the attack mentioned above is only mitigated if the decryption algorithm Dec outputs an empty string ϵ for an incomplete message that puts the decryption oracle ahead of the encryption oracle. This means that the current definition given for $\mathsf{IND\text{-}sfCFA}$ is not necessarily the most natural definition for defining confidentiality in the fragmentation setting.

3.9 A Critique of the Active Boundary Hiding Security Property

The ciphertext fragmentation model uses suppression of output to capture stateful security. This works well in theory and gives a somewhat clean model to define and prove results in. However, does suppression necessarily reflect the real world? The answer is (of course): not very well. It is not possible to assume that real-world applications suppress output. Below, we discuss the consequence of this in relation to the boundary hiding security property. Namely, in some types of application, it is *not* possible to achieve the active boundary hiding notion $\mathsf{BH\text{-}sfCFA}$ in practice, while it, nevertheless, might be able to be proven in theory.

Consider a service that queries a remote database with SQL commands. The remote database will execute the query and immediately transmit the result back to the client. In such an application, active boundary hiding is beneficial, hiding the lengths of the commands and making traffic analysis harder. But even using the InterMAC encryption scheme IM (presented in Chapter 7), an active attacker would be able to tell apart ciphertext boundaries by submitting data to the remote database byte-by-byte and observing when the database responds. In the rest of this section, we will refer to such applications as being *reactive*, i.e., reactive applications are those that produce observable behaviour on, say, a network after decrypting a single ciphertext.

In general, to attack such a reactive application, an adversary can abuse ciphertext fragmentation: submit ciphertext fragments byte-by-byte until a reaction is observed. By keeping track of how many bytes have been submitted, the adversary can infer the length of the ciphertext to which there has been a reaction. This simple, yet fully practical, attack makes it possible for an adversary to delineate ciphertext boundaries, apparently breaking BH-sfCFA for any encryption scheme supporting ciphertext fragmentation. In particular, this would mean that IM presented in Chapter 7 does not meet BH-sfCFA for such applications.

The above discussion highlights a discrepancy between theory and practice concerning the boundary hiding security notions defined in Section 3.6. In theory, there would not be any observable reaction to the legitimate ciphertexts used in the attack, because the decryption oracle suppresses all in-sync decryption output. Thus IM *is* BH-sfCFA secure. In reality, however, the adversary can obtain useful information about ciphertext boundaries by simply observing the network and taking advantage of ciphertext fragmentation. This disjunction between theory and practice seems to be isolated to the definition of active boundary hiding, and we believe it does not affect the usefulness of suppressing output to define other security properties such as confidentiality modelling use-cases (e.g. data transfer over SSH).

3.10 Related Security Models

In this section, we review literature for related security models comparing them to the ciphertext fragmentation model.

3.10.1 Stateful Symmetric Encryption

It is possible to extend the notions IND-CCA and INT-CTXT to capture replay and reordering attacks by an adversary. This was done by Bellare et al. [20, 21]. Their principal motivation was SSH, specifically the SSH encryption schemes that use CBC-mode. They defined two new security notions IND-sfCCA and INT-sfCTXT, pioneering the in-sync/out-of-sync technique also used to define security notions in the ciphertext fragmentation model. The new stateful security notions are constructed in such a way that they imply their corresponding non-stateful notions. The new notions can be seen as a precursor to the ciphertext fragmentation security notions, and the latter are undoubtedly inspired by the former. However, Bellare et al.'s notions only consider the atomic setting. Hence, their results can not be carried over to the ciphertext fragmentation model.

Interestingly, Bellare et al. also proved a version of Theorem 1 for their two new notions. That is, if a scheme meets IND-CPA and INT-sfCTXT then the scheme also meets IND-sfCCA. Similar results have not been proven in the ciphertext fragmentation model, but have been proven for the stream-based channel model [60], which we discuss in Section 3.10.4.

3.10.2 Formal Security Treatment of SSH Encryption Schemes in CTR-mode

Paterson and Watson [114] constructed an SSH inspired model and defined security notions specifically tailored to capture security properties of SSH encryption schemes that use CTR-mode (as implemented in OpenSSH). Their specific focus leads to a number of differences compared to the ciphertext fragmentation model. Firstly, the confidentiality notion [114, Definition 2] (based on the stateful confidentiality notion by Bellare et al. described in Section 3.10.1) includes parameters specific to SSH such as the length field, sequence number and buffer scheme. These parameters are an embedded part of capturing the desired security properties, and as a consequence, their model does not generalise to any other protocols than SSH. Secondly, while the ciphertext fragmentation model allows decryption to continue after a failure, Paterson and Watson require that all subsequent decryption calls must fail (which is indeed the correct behaviour when modelling SSH). This is also an embedded part of their notion of security. Finally, the ciphertext fragmentation model defines several notions that go beyond confidentiality, the main focus of Paterson and Watson.

3.10.3 On-line Symmetric Encryption

The concept of an online symmetric encryption scheme (referred to as an *online cipher* in the literature) has been an active research topic for the past 20 years. The first definition of an online cipher appears in work by Bellare et al. [15], that defines the security of an online cipher to be a function that is indistinguishable from an online random permutation. The term *online* refers to the fact that encryption of the ith plaintext block must only depend on the first i plaintext blocks. This allows an online cipher to output ciphertext prior to having made an entire pass over the plaintext. That is, at a high level, the online setting considers a continuous plaintext and ciphertext where each plaintext block is encrypted to a ciphertext block.

A similar concept to online ciphers, coined *blockwise-adaptive attackers*, were defined in [90] and then morphed into the language of online ciphers in [63, 65, 64]. This setting consider attackers that can adaptively inject message blocks into an encryption process and retrieve ciphertext blocks from a partial message encryption. In these works, it is proved that an adversary equipped with such powers renders several encryption modes insecure in the *Find-Then-Guess* security model [17]. Namely, this is shown for encryption modes CBC, GEM [51] and IACBC [91].

Boldyreva and Taesombut [38] modified the work of Bellare et al. to require constant memory and latency, by only permitting the encryption of the ith block of plaintext to depend on the ith plaintext block, $(i-1)$th plaintext block, and the $(i-1)$th ciphertext block. In addition, they defined new notions, which they claimed to capture the "strongest possible achievable confidentiality notion taking into account blockwise-adaptable attackers" [38, page 3]. This work was later complemented by Rogaway et al. [124]. They define an online cipher to be a deterministic, length-preserving block cipher $E\colon (\{0,1\}^n)^+ \to (\{0,1\}^n)^+$, where the ith block of ciphertext only depends on the first i blocks of plaintext, extend the online cipher paradigm to allow variable-length messages, and use this to construct online ciphers from tweakable block ciphers [102]. Furthermore, [81] critiques the original security notion of an online cipher (OAE1 in their language). Instead, they present a new security notion (OAE2), which, in their words, gives "a more faithful delivery on the promise of achieving best-possible security for an online authenticated symmetric encryption scheme". [81, Figure 1] compares OAE1 and OAE2.

Additional works [12, 11, 61, 1, 62, 10, 9, 33] developed and analysed different aspects of online ciphers.

The ciphertext fragmentation model is distinguishable from the online cipher model in some key areas. Firstly, the ciphertext fragmentation model only models atomic encryption of messages and does not give blockwise-adaptive-style powers to an adversary. Secondly, the ciphertext fragmentation model aims to model the non-atomic behaviour of ciphertext delivery and does this by abstracting away the specific low-level functionality of the encryption/decryption operation; there is no assumption of specific block-wise encryption/decryption behaviour. This seeks to reflect how many transport layers in secure communications protocols operate. Thirdly, while there exist many works on online ciphers dealing with passive adversaries, there is not much work considering active attackers. The ciphertext fragmentation model emphasises the existence of active adversaries. Finally, the ciphertext fragmentation model defines a number of additional security notions that are not defined in the online cipher setting. Namely, the boundary hiding notion and the denial-of-service security notion.

3.10.4 Stream-based Channel

In [60] Fischlin et al. construct a model that attempts to simulate the distinctive behaviour of streaming APIs. Such APIs do not receive discrete plaintexts or ciphertexts, but instead receive plaintext/ciphertext fragments (in the same way as we define ciphertext fragments in the ciphertext fragmentation model). In addition, the sending side may arbitrarily buffer input, only forwarding at its discretion. This sort of behaviour is distinctive of, for example, TCP-like socket interfaces, the sort of interfaces offered by e.g. TLS and QUIC [86].

To model streaming APIs, Fischlin et al. define a model capturing a stream-based channel which is composed of three algorithms: an init algorithm init, a send algorithm send and a receive algorithm recv. The send algorithm can be viewed as the encryption algorithm, and the receive algorithm can be viewed as the decryption algorithm. The send algorithm is special in that it allows input of plaintexts but does not necessarily output any ciphertext. Instead, output is controlled by a flush flag (a parameter to the send algorithm). This achieves two goals: it views plaintexts as units in a stream and not atomic/discrete units, and allows the sending algorithm send to process data at its discretion.

The receive algorithm is similar in nature to the decryption algorithm in the ciphertext fragmentation model and must handle fragments that do not necessarily represent an atomic plaintext. As with the input to the send algorithm, Fischlin et al. view the input to the receive algorithm as a stream of ciphertexts. The special features of the send and receive algorithms are nicely illustrated in [60, Figure 2].

The stream-based channel model in [60] has striking similarities to the ciphertext fragmentation model but with some essential differences. Firstly, the ciphertext fragmentation model views plaintexts as discrete units, and the encryption algorithm does not have any discretion on when to output ciphertext; this must happen after processing the plaintext input. This difference, of course, separates the security results obtained in each model. Both models, however, define analogous notions for confidentiality and authenticity. This leads us to the second major difference between the two models. The ciphertext fragmentation model defines several security notions that go beyond confidentiality and authenticity. It is possible, however, that future work would define similar notions for the stream-based channel model.

Bhargavan et al. [32] also define stream-based security notions that are similar in nature to Fischlin et al.'s security notions, capturing security properties as types in a programming language. An adversary is modelled as another program interacting with the code for the send and receive algorithms. This work was extended in [56], to capture security properties of TLS 1.3 [119]. This work aimed to make formal analysis techniques easier to apply and presents a direction that might be beneficial in the future as formal analysis methods continue to develop.

Chapter 4

The SSH Ecosystem on the Internet

In this chapter, we present a study of the SSH ecosystem on the Internet. We focus on the mix of different SSH implementations, versions supported and preferred SSH encryption scheme. Our search is entirely restricted to SSH servers with a public IP address reachable on the Internet.

4.1 Ethical Considerations

When performing Internet-wide scanning it is practically impossible to request permission from all owners of the IP addresses scanned. One noticeable ethical risk of doing Internet-wide scanning is that our scan risks slow down, or worse, bring down, the target system. Simultaneously, there is also the risk of decreasing performance of the local system performance the scan, which might be problematic if you are using e.g. the university Internet connection. Another noticeable risk is the risk of bad sentiment towards the entity from which the scan originates.

We were performing our scans from a host located on the Royal Holloway, University of London, network. Taking the above risks into account, we (lead by Martin Albrecht) obtained official approval from the IT/IT-security department before commencing our scans. To reduce the risk of slowing down target and source systems, we limited the number of packets to 10000 / second.

4.2 Data Set

Our data set consists of data collected from three active scans we ran on the entire IPv4-address-space using ZGrab/ZMap [57]. For every reachable IP, we attempted to

Scan	SSH server count	SSH servers with scheme data count
scan 2015-12	17,747,039	17,697,767
scan 2016-01	17,194,797	17,146,588
scan 2019-11	14,762,396	14,718,379

Table 4.1: The second column displays the number of SSH servers found in each scan. The third column displays the number of SSH servers found in each scan for which the supported SSH encryption schemes could also be collected.

establish an SSH connection on port 22 (the default SSH port). If an SSH server was listening on port 22, we collected metadata from the initial key establishment and terminated the connection. The metadata collected comprised of the server banner, containing information about the software and version of the SSH server, as well as SSH encryption schemes offered by the server. We count a server as an SSH server if we were able to collect the banner.

The scans were conducted from November 11, 2015, to December 1, 2015, from January 22, 2016, to January 27, 2016, and from November 4, 2019, to November 13, 2019, henceforth referred to as the 2015 scan, 2016 scan and the 2019 scan, respectively. We found about 2^{24} (approximately 16 million) servers in the first two scans and about two million fewer servers in the last scan. For some servers indicating SSH support, the connection terminated prematurely. This was either due to a connection timeout or an error during the handshake. Table 4.1 shows the number of servers for which we were able to collect the SSH banner, and the number of servers where the supported SSH encryption schemes could also be determined.

It is important to note that our data collected does not necessarily accurately reflect the actual SSH encryption scheme negotiated at the end of the key establishment or the software used by the SSH server. In SSH, SSH encryption schemes are negotiated based on the preferences of the client, not the server: the first option from the client's list that is also on the server's list is used. Hence, similarly to recent studies of algorithm deployment in TLS (e.g. [97]), it is difficult to establish what schemes are actually used to protect traffic in flight, because only servers and their preferences can be easily queried. In addition, servers might be listening on other ports than 22 as a protection measure. Still, the data presented in this chapter should give a rough estimation of configurations in the wild since the most prominent servers — OpenSSH and Dropbear — share code and default configurations between server and client.

The server software is determined by an initial banner that is sent by the server. However, this banner can easily be modified by the server administrator without any available method that can be used to verify its correctness. The server banner is regularly modified to reflect the context in which the server is used or to hide which software the server is using. In Section 4.6.4, we will report on servers reporting *AWS_SFTP_1.0* in the banner, which may be a case of the former. In Section 4.6.3, we will report on servers reporting *XXXX* in the banner, which may be a case of the latter.

4.3 SSH Deployment Statistics

In this section, we report on SSH deployment statistics, focussing on the mix of different SSH implementations and versions supported by SSH servers.

4.3.1 SSH Implementations and Versions Statistics in the 2019 Scan

As Table 4.2 shows, amongst those servers our 2019 scan identified, those self-reporting as OpenSSH dominate, with version 7.4 (*OpenSSH_7.4*) being the most popular version. The most popular non-OpenSSH server is Dropbear running version 2016.74 (*dropbear_2016.74*). Overall, the landscape is dominated by OpenSSH with approximately 86% in the 2019 scan reported as some version of OpenSSH. In contrast, the second largest software family, Dropbear, makes up approximately 7% of the total number of servers. There is a long tail of servers reporting as software not belonging to the OpenSSH or Dropbear families. Among the top 35 servers these are *Cisco-1.25*, *ROSSSH*, *XXXX* and *AWS_SFTP_1.0*. We investigate these families further in Section 4.6.

A total of 111,635 unique implementation identifiers were reported by all servers in the 2019 scan. 99,057 identifiers were only reported once, which amounts to approximately 89% of all identifiers. Only 426 identifiers were reported more than 100 times, approximately 3.5% of all identifiers.

software	scan 2019–11		scan 2016–01		scan 2015–12	
OpenSSH_7.4	2,847,400	(19.3%)	0	(0.0%)	0	(0.0%)
OpenSSH_7.2p2	1,921,416	(13.1%)	0	(0.0%)	0	(0.0%)
OpenSSH_5.3	1,714,255	(11.6%)	2,108,738	(12.3%)	2,133,772	(12.0%)
OpenSSH_7.6p1	1,545,127	(10.5%)	0	(0.0%)	0	(0.0%)
OpenSSH_7.4p1	896,419	(6.1%)	0	(0.0%)	0	(0.0%)
OpenSSH_6.6.1p1	646,817	(4.4%)	1,198,987	(7.0%)	1,124,914	(6.3%)
OpenSSH_6.7p1	544,952	(3.7%)	261,867	(1.5%)	21,3843	(1.2%)
OpenSSH_6.6.1	490,104	(3.3%)	338,787	(2.0%)	252,856	(1.4%)
dropbear_2016.74	310,760	(2.1%)	0	(0.0%)	0	(0.0%)
OpenSSH_7.5	273,757	(1.9%)	0	(0.0%)	0	(0.0%)
Cisco-1.25	232,989	(1.6%)	0	(0.0%)	0	(0.0%)
OpenSSH_7.9p1	186,151	(1.3%)	0	(0.0%)	0	(0.0%)
OpenSSH_6.0p1	178,902	(1.2%)	554,295	(3.2%)	573,634	(3.2%)
dropbear_2015.67	152,645	(1.0%)	35,989	(0.2%)	30,277	(0.2%)
OpenSSH_4.3	150,614	(1.0%)	1,716	(<0.1%)	1,607	(<0.1%)
dropbear	147,936	(1.0%)	11,369	(0.1%)	11,121	(0.1%)
ROSSSH	142,973	(1.0%)	345,916	(2.0%)	333,992	(1.9%)
OpenSSH_8.0	113,282	(0.8%)	0	(0.0%)	0	(0.0%)
OpenSSH_7.9	102,687	(0.7%)	0	(0.0%)	0	(0.0%)
OpenSSH_5.9p1	96,510	(0.7%)	467,899	(2.7%)	500,975	(2.8%)
dropbear_0.46	90,040	(0.6%)	301,913	(1.8%)	335,425	(1.9%)
OpenSSH_6.4	65,431	(0.4%)	78,040	(0.5%)	79,468	(0.4%)
OpenSSH_7.2	62,911	(0.4%)	0	(0.0%)	0	(0.0%)
OpenSSH_5.5p1	56,917	(0.4%)	262,367	(1.5%)	272,990	(1.5%)
OpenSSH_5.1	53,741	(0.4%)	86,338	(0.5%)	44,170	(0.2%)
XXXX	51,708	(0.4%)	36,654	(0.2%)	35,857	(0.2%)
OpenSSH_7.8	48,860	(0.3%)	0	(0.0%)	0	(0.0%)
OpenSSH_6.2	47,204	(0.3%)	255,088	(1.5%)	288,710	(1.6%)
dropbear_2017.75	42,978	(0.3%)	0	(0.0%)	0	(0.0%)
dropbear_2014.63	41,740	(0.3%)	422,764	(2.5%)	197,353	(1.1%)
OpenSSH_7.6	38,780	(0.3%)	0	(0.0%)	0	(0.0%)
OpenSSH_5.8	38,558	(0.3%)	88,258	(0.5%)	89,144	(0.5%)
AWS_SFTP_1.0	38,539	(0.3%)	0	(0.0%)	0	(0.0%)
dropbear_2012.55	32,756	(0.2%)	82,920	(0.5%)	86,675	(0.5%)
dropbear_0.50	30,750	(0.2%)	41,222	(0.2%)	44,107	(0.2%)

Table 4.2: Deployment statistics for SSH servers in 2015, 2016 and 2019 scans.

4.3.2 Evolution in SSH Implementations and Versions

When only considering the 2016 and 2015 scans, the picture changes completely compared to Section 4.3.1. As Table 4.3 shows, amongst those servers our scan identified, those self-reporting as *dropbear_2014.66* dominate, with *OpenSSH_5.3* being the second most popular. Overall, the landscape is dominated by OpenSSH and Dropbear servers, with *ROSSSH* being the only noticeable server in the top 35 not belonging to either family. Overall, 56.11% (resp. 57.97%) of all servers reported as some version of Dropbear and 39.22% (resp. 37.17%) as some version of OpenSSH in the 2016 (resp. 2015) scan. In both scans, less than 5% of servers reported as something other than Dropbear or OpenSSH. This number is up in the 2019 scan to 7% overall. In 2019, the most popular Dropbear server is *dropbear_2016.74*, while there are no servers self-reporting as *dropbear_2014.66*.

software	scan 2016–01		scan 2015–12	
dropbear_2014.66	7,229,491	(42.0%)	8,334,758	(47.0%)
OpenSSH_5.3	2,108,738	(12.3%)	2,133,772	(12.0%)
OpenSSH_6.6.1p1	1,198,987	(7.0%)	1,124,914	(6.3%)
OpenSSH_6.0p1	554,295	(3.2%)	573,634	(3.2%)
OpenSSH_5.9p1	467,899	(2.7%)	500,975	(2.8%)
dropbear_2014.63	422,764	(2.5%)	197,353	(1.1%)
dropbear_0.51	403,923	(2.3%)	434,839	(2.5%)
dropbear_2011.54	383,575	(2.2%)	64,666	(0.4%)
ROSSSH	345,916	(2.0%)	333,992	(1.9%)
OpenSSH_6.6.1	338,787	(2.0%)	252,856	(1.4%)
dropbear_0.46	301,913	(1.8%)	335,425	(1.9%)
OpenSSH_5.5p1	262,367	(1.5%)	272,990	(1.5%)
OpenSSH_6.7p1	261,867	(1.5%)	213,843	(1.2%)
OpenSSH_6.2	255,088	(1.5%)	288,710	(1.6%)
dropbear_2013.58	236,409	(1.4%)	249,284	(1.4%)
dropbear_0.53	217,970	(1.3%)	213,670	(1.2%)
dropbear_0.52	132,668	(0.8%)	136,196	(0.8%)
OpenSSH	110,602	(0.6%)	108,520	(0.6%)
OpenSSH_5.8	88,258	(0.5%)	89,144	(0.5%)
OpenSSH_5.1	86,338	(0.5%)	44,170	(0.2%)
OpenSSH_5.3p1	84,559	(0.5%)	89,780	(0.5%)
OpenSSH_7.1	83,793	(0.5%)	73,193	(0.4%)
dropbear_2012.55	82,920	(0.5%)	86,675	(0.5%)
ARRIS_0.50	81,744	(0.5%)	156,489	(0.9%)
OpenSSH_5.1p1	80,207	(0.5%)	83,964	(0.5%)
OpenSSH_6.4	78,040	(0.5%)	79,468	(0.4%)
OpenSSH_6.6.1_hpn13v11	60,696	(0.4%)	55139	(0.3%)
OpenSSH_5.9	56,268	(0.3%)	58,614	(0.3%)
homepl	56,225	(0.3%)	58,359	(0.3%)
OpenSSH_5.2	50,136	(0.3%)	53,199	(0.3%)
RomSShell_4.31	42,298	(0.2%)	43,890	(0.2%)
dropbear_0.50	41,222	(0.2%)	44,107	(0.2%)
OpenSSH_6.6	41,024	(0.2%)	39,573	(0.2%)
OpenSSH_6.9p1	39,909	(0.2%)	22,218	(0.1%)
lancom	39,639	(0.2%)	39,661	(0.2%)

Table 4.3: Deployment statistics for SSH servers in 2016 and 2015 scans.

4.4 SSH Encryption Scheme Statistics

In this section, we report on SSH deployment statistics, focussing on the SSH encryption schemes preferred by SSH servers.

4.4.1 Default SSH Encryption Schemes Preferences

To negotiate an SSH encryption scheme, the client and server each exchange two lists of algorithms: a list of symmetric encryption algorithms and a list of message authentication code algorithms. The algorithm order in each list matters: algorithms appear in descending order of preference, the first algorithm being the most preferred. As mentioned earlier, the client decides which specific algorithms are selected.

SSH software normally comes with predefined lists of algorithms that the client and server use. However, the lists can be modified locally. Hence, the client and server lists might differ even when the client and server use the same software and software version.

OpenSSH

The current default list of symmetric encryption schemes and message authentication codes in OpenSSH (as of version 8.1) is the following for both client and server:

Symmetric encryption algorithms: {`"chacha20-poly1305@openssh.com"`, `"aes128-ctr"`,`"aes192-ctr"`,`"aes256-ctr"`,`"aes128-gcm@openssh.com"`, `"aes256-gcm@openssh.com"`}

MAC algorithms: {`"umac-64-etm@openssh.com"`, `"umac-128-etm@openssh.com"`,`"hmac-sha2-256-etm@openssh.com"`, `"hmac-sha2-512-etm@openssh.com"`,`"hmac-sha1-etm@openssh.com"`, `"umac-64@openssh.com"`,`"umac-128@openssh.com"`,`"hmac-sha2-256"`, `"hmac-sha2-512"`,`"hmac-sha1"`}

OpenSSH has seen a steady trend of deprecating and removing support for older algorithms and, at the same time, implementing support for newer and more modern algorithms. For example, of 16 algorithms in the lists above, 10 algorithms are OpenSSH-specific algorithms (indicated by the suffix `@openssh.com`). In addition, OpenSSH strongly favours Encrypt-then-MAC SSH encryption schemes over the standard SSH option of Encrypt-and-MAC.

Dropbear

The current default list in Dropbear (as of version 2019.78) is the following for both client and server:

Symmetric encryption algorithms: {`"aes128-ctr"`,`"aes192-ctr"`,`"aes256-ctr"`, `"twofish256-ctr"`,`"twofish128-ctr"`,`"aes128-cbc"`,`"aes256-cbc"`,`"twofish256-cbc"`, `"twofish128-cbc"`,`"3des-ctr"`,`"3des-cbc"`,`"blowfish-cbc"`}

MAC algorithms : {`"hmac-sha1-96"`,`"hmac-sha1"`,`"hmac-sha2-256"`, `"hmac-sha2-512"`,`"hmac-md5"`}

Twofish algorithms must be enabled at compile-time, but when enabled, they enter the list at the preference shown above. Dropbear has not had the same drive towards newer cryptographic algorithms as OpenSSH and has stayed with originally recommended algorithms. Dropbear does recommend CTR-mode over CBC-mode, which, considering later chapters in this thesis, is a good choice (cf. Chapter 5). An option to disable CBC-mode ciphers was added in Dropbear version 2015.67 (released 28/1/2015).

4.4.2 SSH Encryption Scheme Diversity

We show the preferred combinations of encryption and MAC algorithms found in our third (November 2019) scan for all OpenSSH servers, all Dropbear servers, and all servers overall, in Table 4.4, Table 4.5, and Table 4.6, respectively. In total, we saw 253 different combinations as first preference at one or more SSH servers and 178 different combinations for OpenSSH. Of course, many of these combinations are used by tiny numbers of servers, but it is still noteworthy that there is so much diversity in the algorithm combinations for SSH. These numbers have increased compared to the 2016 scan where we only saw 199 different combinations as first preference overall and 155 combinations for OpenSSH specifically. The Dropbear family contained the next-most different combinations, 36, almost five times less than the OpenSSH family. However, for Dropbear, the number of combinations is also up compared to the 2016 scan where we only saw a total of 16 different combinations.

4.4.3 Evolution in SSH Encryption Scheme Preferences

In Table 4.7, we show the evolution in preferred SSH encryption schemes between the 2016 and 2019 scan for the servers self-reporting as belonging to the OpenSSH family. The most popular SSH encryption scheme is SSH-ChaCha20-Poly1305 making up 57.8% of preferred schemes over all OpenSSH servers. This is in stark contrast to 2016, were the number of OpenSSH servers preferring SSH-ChaCha20-Poly1305 was only a tiny 1.7%.

Likewise, there have been a shift in the preferred SSH encryption scheme for servers self-reporting as belonging to the Dropbear family. Table 4.8 shows that the shift is mainly in diversity. In 2016, a large majority of Dropbear servers preferred the SSH encryption scheme SSH-AES128-CTR-HMAC-SHA1-96 (90.4%), while in 2019 there

encryption and mac algorithm	count	
`chacha20-poly1305@ + umac-64-etm@`	7,369,760	(57.8%)
`aes128-ctr + hmac-md5`	2,088,201	(16.4%)
`aes128-ctr + hmac-md5-etm@`	1,216,840	(9.5%)
`aes128-ctr + umac-64-etm@`	705,031	(5.5%)
`aes128-cbc + hmac-md5`	274,881	(2.2%)
`aes128-gcm@ + umac-64-etm@`	190,290	(1.5%)
`chacha20-poly1305@ + hmac-sha2-512-etm@`	188,289	(1.5%)
`aes256-ctr + hmac-sha2-512`	163,732	(1.3%)
`aes128-ctr + hmac-sha1`	109,631	(0.9%)
`aes128-ctr + hmac-sha2-256`	91,182	(0.7%)
`aes256-gcm@ + hmac-sha2-512-etm@`	80,317	(0.6%)
`aes256-gcm@ + hmac-sha2-256-etm@`	50,157	(0.4%)
`aes128-ctr + hmac-ripemd160`	21,837	(0.2%)
`aes128-cbc + hmac-sha1`	16,958	(0.1%)
`chacha20-poly1305@ + hmac-md5-etm@`	14,204	(0.1%)
`chacha20-poly1305@ + umac-128-etm@`	13,788	(0.1%)
`chacha20-poly1305@ + hmac-sha1`	11,618	(0.1%)
`aes128-gcm@ + hmac-sha2-512-etm@`	11,037	(0.1%)
`aes128-cbc + hmac-sha2-512-etm@`	8,293	(0.1%)
`aes256-ctr + hmac-sha1`	7,920	(0.1%)

Table 4.4: Encryption and MAC algorithm preferences for OpenSSH servers in the 2019 scan. `@openssh.com` is abbreviated to `@`.

are two popular preferred SSH encryption schemes: SSH-AES128-CTR-HMAC-SHA1 (36.6%) and SSH-AES128-CTR-HMAC-SHA1-96 (33.4%). The two schemes are identical, except the latter truncates the MAC tag to 96 bits. The last entry in Table 4.8 shows that 38 Dropbear servers prefer SSH-ChaCha20-Poly1305. Since Dropbear does not natively support this scheme, these servers are likely OpenSSH servers self-reporting as a Dropbear server, or a local patch has been applied implementing support for the scheme, without pushing it upstream. In total, we saw 201 servers self-reporting as Dropbear but also preferring an SSH encryption scheme that use neither CBC-mode or CTR-mode.

4.5 CBC-mode Vulnerabilities

In Chapter 5, we will recall the Albrecht-Paterson-Watson attack (denoted by *Attack APW* in this chapter) [5], and present three new attacks against CBC-mode SSH encryption schemes in OpenSSH. These attacks will be denoted *Attack one*, *Attack*

encryption and mac algorithm	count	
aes128-ctr + hmac-sha1	383,348	(36.6%)
aes128-ctr + hmac-sha1-96	349,588	(33.4%)
aes128-ctr + hmac-sha2-256	138,552	(13.2%)
3des-cbc + hmac-sha1	117,881	(11.3%)
aes128-cbc + hmac-sha1-96	42,550	(4.1%)
3des-ctr + hmac-sha1	5,734	(0.5%)
aes128-cbc + hmac-sha1	2,781	(0.3%)
twofish128-cbc + hmac-sha1-96	2,232	(0.2%)
3des-ctr + hmac-sha1-96	1,792	(0.2%)
aes256-ctr + hmac-sha1	1,372	(0.1%)
3des-cbc + hmac-sha1-96	746	(0.1%)
aes256-ctr + hmac-sha1-96	333	(<0.1%)
aes256-ctr + hmac-sha2-256	186	(<0.1%)
aes128-ctr + hmac-sha2-256-etm@	39	(<0.1%)
chacha20-poly1305@ + hmac-sha1-96	38	(<0.1%)
aes192-ctr + hmac-sha1-96	36	(<0.1%)
aes256-ctr + hmac-sha2-256-etm@	32	(<0.1%)
aes192-ctr + hmac-sha2-256-etm@	32	(<0.1%)
aes192-ctr + hmac-sha1	30	(<0.1%)
chacha20-poly1305@ + hmac-sha2-256	29	(<0.1%)

Table 4.5: Encryption and MAC algorithm preferences for Dropbear servers in the 2019 scan. @openssh.com is abbreviated to @.

two and *Attack three*. Below, we will highlight how many servers are vulnerable to these attacks. A server is considered to be vulnerable if the server prefers a CBC-mode scheme, running in Encryption-and-MAC mode, and the self-reported server software version is a version for which at least one of the attacks are applicable.

4.5.1 OpenSSH

Referring forward to Chapter 5, we list the vulnerable OpenSSH software versions for which each attack is applicable:

Attack APW: All OpenSSH versions up to, and including, version 5.1.

Attack one: OpenSSH versions 5.2–5.9, 6.0–6.9 and 7.0–7.4.

Attack two: OpenSSH versions 5.2–5.9, 6.0–6.9 and 7.0–7.2.

Attack three: OpenSSH versions 7.3–7.9 and 8.0-current.

It is apparent, that no matter what version of OpenSSH preferring a CBC-mode

encryption and mac algorithm	count	
chacha20-poly1305@ + umac-64-etm@	7,505,362	(51.0%)
aes128-ctr + hmac-md5	2,125,031	(14.4%)
aes128-ctr + hmac-md5-etm@	1,221,387	(8.3%)
aes128-ctr + hmac-sha1	739,385	(5.0%)
aes128-ctr + umac-64-etm@	705,261	(4.8%)
aes128-ctr + hmac-sha1-96	371,891	(2.5%)
aes128-cbc + hmac-md5	333,974	(2.3%)
aes128-ctr + hmac-sha2-256	268,679	(1.8%)
aes128-gcm@ + umac-64-etm@	190,326	(1.3%)
aes128-cbc + hmac-sha1	189,279	(1.3%)
chacha20-poly1305@ + hmac-sha2-512-etm@	188,364	(1.3%)
3des-cbc + hmac-sha1	168,515	(1.1%)
aes256-ctr + hmac-sha2-512	167,549	(1.1%)
aes256-gcm@ + hmac-sha2-512-etm@	80,328	(0.5%)
aes128-cbc + hmac-sha1-96	50,847	(0.3%)
aes256-gcm@ + hmac-sha2-256-etm@	50,173	(0.3%)
aes256-ctr + hmac-sha2-256	43,641	(0.3%)
aes256-ctr + hmac-sha1	23,307	(0.2%)
aes128-ctr + hmac-ripemd160	21,839	(0.1%)
aes192-cbc + hmac-sha1	19,547	(0.1%)

Table 4.6: Encryption and MAC algorithm preferences for all servers in the 2019 scan. `@openssh.com` is abbreviated to `@`.

scheme is vulnerable to at least one of the attacks. A total of 333,376 OpenSSH servers prefer a CBC-mode scheme, amounting to 2.6% of all OpenSSH servers. All these servers are likely to be vulnerable to one of the attacks presented in Chapter 5. Table 4.9 shows the number of vulnerable OpenSSH servers found in the 2019 and 2016 scans, divided into the different type of attacks.

4.5.2 Dropbear

Dropbear did not implement a counter-measure to the CBC-mode attack of [5]. Instead, *dropbear_0.52* (released 12/11/2008) added support for CTR-mode and made it the default. Hence, any Dropbear server preferring CBC-mode is vulnerable to a variant of the attack from [5] (described in Section 5.1.1). This includes any version of Dropbear prior to version 0.52.

Table 4.10 shows the number of vulnerable Dropbear servers in the 2016 and 2019 scans. We found substantially fewer vulnerable servers in the 2016 scan compared

encryption and mac algorithm	scan 2019–11		scan 2016–01	
`chacha20-poly1305@ + umac-64-etm@`	7,369,760	(57.8%)	115,526	(1.7%)
`aes128-ctr + hmac-md5`	2,088,201	(16.4%)	3,877,790	(57.7%)
`aes128-ctr + hmac-md5-etm@`	1,216,840	(9.5%)	2,010,936	(29.9%)
`aes128-ctr + umac-64-etm@`	705,031	(5.5%)	331,014	(4.9%)
`aes128-cbc + hmac-md5`	274,881	(2.2%)	161,624	(2.4%)
`aes128-gcm@ + umac-64-etm@`	190,290	(1.5%)	110	(0.0%)
`chacha20-poly1305@ + hmac-sha2-512-etm@`	188,289	(1.5%)	8179	(0.1%)
`aes256-ctr + hmac-sha2-512`	163,732	(1.3%)	17,897	(0.3%)
`aes128-ctr + hmac-sha1`	109,631	(0.9%)	68,027	(1.0%)
`aes128-ctr + hmac-sha2-256`	91,182	(0.7%)	7,773	(0.1%)
`aes256-gcm@ + hmac-sha2-512-etm@`	80,317	(0.6%)	28,019	(0.4%)
`aes256-gcm@ + hmac-sha2-256-etm@`	50,157	(0.4%)	5	(<0.1%)
`aes128-ctr + hmac-ripemd160`	21,837	(0.2%)	10621	(0.2%)
`aes128-cbc + hmac-sha1`	16958	(0.1%)	11082	(0.2%)
`chacha20-poly1305@ + hmac-md5-etm@`	14204	(0.1%)	185	(<0.1%)

Table 4.7: Comparison of preferred encryption and MAC algorithms for OpenSSH servers, between the 2016 and 2019 scans. `@openssh.com` abreviated to `@`.

to the 2019 scan. However, we also found significantly more servers in the 2016 scan. Taking this into account, we found that 15.8% and 8.4% of the Dropbear servers in the 2019 and 2016 scans, respectively, were vulnerable. Hence, we found proportionally more vulnerable servers in 2019 than in 2016.

4.6 Less Frequent SSH Software Identifiers

In this section, we explore some of the less popular SSH software options discovered in the 2019 scan. Specifically, we investigate *Cisco*, *ROSSSH*, *XXXX* and *AWS_SFTP_1.0*. In turns out that 3 out 4 of these are likely software derived from either OpenSSH or Dropbear.

4.6.1 Cisco

Cisco SSH software is part of the Cisco IOS (Internetwork Operating System), used on many Cisco routers and switches.[1]

A total of 232,989 and 10,301 servers self-reported as *Cisco-1.25* or *Cisco-2.0*, respectively. A total of 132,164 servers accept SSH protocol version 1 connections. Of these servers, 126,036 support SSH protocol version 1.9, which is a generally accepted method to indicate that the server supports both SSH protocol versions 1

[1] `https://www.cisco.com/c/en_uk/products/ios-nx-os-software/index.html`

encryption and mac algorithm	scan 2019–11		scan 2016–01	
aes128-ctr + hmac-sha1	383,348	(36.6%)	62,465	(0.6%)
aes128-ctr + hmac-sha1-96	349,588	(33.4%)	8,724,863	(90.4%)
aes128-ctr + hmac-sha2-256	138,552	(13.2%)	36,150	(0.4%)
3des-cbc + hmac-sha1	117,881	(11.3%)	321,492	(3.3%)
aes128-cbc + hmac-sha1-96	42,550	(4.1%)	478,181	(5.0%)
3des-ctr + hmac-sha1	5,734	(0.5%)	7,058	(0.1%)
aes128-cbc + hmac-sha1	2,781	(0.3%)	14,477	(0.2%)
twofish128-cbc + hmac-sha1-96	2,232	(0.2%)	0	(0.0%)
3des-ctr + hmac-sha1-96	1,792	(0.2%)	175	(<0.1%)
aes256-ctr + hmac-sha1	1,372	(0.1%)	2	(<0.1%)
3des-cbc + hmac-sha1-96	746	(0.1%)	2,043	(<0.1%)
aes256-ctr + hmac-sha1-96	333	(<0.1%)	0	(0.0%)
aes256-ctr + hmac-sha2-256	186	(<0.1%)	0	(0.0%)
aes128-ctr + hmac-sha2-256-etm@	39	(<0.1%)	0	(0.0%)
chacha20-poly1305@ + hmac-sha1-96	38	(<0.1%)	0	(0.0%)

Table 4.8: Comparison of preferred encryption and MAC algorithms for Dropbear servers, between the 2016 and 2019 scans. `@openssh.com` abreviated to `@`.

Attack	scan 2019–11	scan 2016–01
Attack APW	221,552	166,572
Attack one	80,228	Not applicable
Attack two	68,357	Not applicable
Attack three	37,435	Not applicable

Table 4.9: Number of vulnerable OpenSSH servers in the 2016 and 2019 scans for each attack presented in Chapter 5.

and 2, while 6,128 servers only support SSH protocol version 1. 10 servers reported that they support SSH protocol version 2.99 (which does not exist). This is likely caused by a bug in the Cisco software.[2]

The vast majority of Cisco SSH servers prefer either CBC-mode or CTR-mode SSH encryption schemes. 134,242 (54.7%) servers prefer the former, while 109,823

[2] https://community.cisco.com/t5/vpn-and-anyconnect/ssh-version-2-99/td-p/1066921

Attack	scan 2019–11	scan 2016–01
Attack APW	166,234	816,359

Table 4.10: Number of vulnerable Dropbear servers found in the 2016 and 2019 scans.

(44.8%) servers prefer the latter. A subset of Cisco servers also supports the scheme SSH-ChaCha20-Poly1305, with 970 servers having it as their preferred SSH encryption scheme. This might indicate that the Cisco SSH server software is derived from OpenSSH. If this is the case, more than half (54.7%) the servers self-reporting as Cisco servers might be vulnerable to one of the attacks presented in Chapter 5.

4.6.2 ROSSSH

ROSSSH is an SSH implementation included in the RouterOS software by MikroTik.[3] A small number of ROSSSH servers support OpenSSH-specific SSH encryption schemes. However, the changelogs for RouterOS[4] do not indicate that such schemes have officially been added to ROSSSH. It is therefore unclear whether ROSSSH is derived from OpenSSH. In addition, a few ROSSSH servers prefer the encryption algorithms `twofish-cbc` and `twofish256-cbc`. We found zero OpenSSH servers that prefer any of those two algorithms.

A total of 19,535 (13.6%) of ROSSSH servers prefer an SSH encryption scheme using CBC-mode, while 1123,334 (86.2%) server prefer schemes using CTR-mode. If ROSSSH is derived from either OpenSSH or Dropbear, this would likely leave 13.6% of servers vulnerable to at least one attack presented in Chapter 5.

4.6.3 XXXX

Using the search query `xxxx port:"22"` on `shodan.io`, shows that XXXX SSH servers are located in all parts of the world, with the majority of IPs originating from India. 47,751 (88.1%) XXXX servers support the key exchange algorithm `kexguess2@matt.ucc.asn.au`. This algorithm is a Dropbear-specific algorithm not officially supported by OpenSSH or any other SSH software (to the best of our knowledge). This indicates that the SSH software on XXXX servers is derived from Dropbear.

There are 1,166 (2.1%) XXXX servers preferring CBC-mode SSH encryption schemes. If XXXX is derived from Dropbear, it is likely these servers are vulnerable to the CBC-mode attack against Dropbear described in Section 5.1.1.

[3] https://mikrotik.com/

[4] https://mikrotik.com/download/changelogs

4.6.4 AWS_SFTP_1.0

AWS SFTP[5] is an Amazon web service that allows customers to transfer data to and from the Amazon cloud using the SFTP protocol over SSH. All servers self-reporting as AWS_SFTP_1.0 offer the same algorithm lists, and all prefer the scheme SSH-ChaCha20-Poly1305. None of the servers support CBC-mode SSH encryption schemes.

The algorithm lists returned from AWS_SFTP_1.0 are identical to the default lists described in Section 4.4.1. It is therefore highly likely that AWS_SFTP_1.0 SSH software is derived from a (likely newer) OpenSSH version.

4.7 Noticeable Findings

In this section, we highlight some noticeable findings derived from the observations presented in previous sections.

4.7.1 Age of Actively Used SSH Software

OpenSSH version 5.3 was released on 01/10/2009. Yet, in 2016 it was still the most popular OpenSSH version, 7 years after its release. In 2019 the situation had improved, with OpenSSH version 7.4 (released 19/12/2016) now being the most popular version. That is, the most popular OpenSSH version in 2019 is trailing behind the newest version by only 3 years compared to 7 years in 2016. This is a significant improvement from a cryptographic perspective because newer versions of OpenSSH have been deprecating cryptographically weak algorithms, introducing modern cryptographic algorithms, and improving default settings.

dropbear_2014.66 dominated in the 2016 scan. However, this version completely disappeared in our 2019 scan, where the Dropbear server version *dropbear_2016.74* was the most popular. On the other hand, the 2019 scan found significantly fewer Dropbear servers. The “missing” servers might still be in operation running the older Dropbear version, but blocking inbound scan attempts.

Many operating systems ship with some version of OpenSSH (including Redhat, OSx and OpenBSD). When one of these operating systems are installed on a server, it is

[5] `https://aws.amazon.com/aws-transfer-family`

not likely that the SSH software is updated to the latest version or replaced with other software. Hence, the server defaults to using the SSH software and version provided by the operating system. This could be one of the main reasons why the most popular OpenSSH version is trailing behind the newer versions. This situation also likely arises with Dropbear. Since Dropbear has a small memory footprint, it is a desirable candidate for SSH software in embedded devices[6], and it can be hard to upgrade software on such devices.

This can also explain the high number of old software versions reported in the 2019 scan. We found 43,470 OpenSSH servers running a version lower than 4.0. These versions are at least 15 years old. At the same time, we found 5,904 servers running OpenSSH version 1.x. These versions are 20 years old. The same is true for Dropbear. We found 196,927 Dropbear servers running version 0.x. The oldest version of 0.x (0.28) was released on 06/04/2003, and the newest version of 0.x (0.53.1) was released on 02/03/2011 - a span of approximately 8 years. The Dropbear servers running version 0.x are, therefore, likely to be running software versions that are in the span of 9 to 17 years old. Generally, however, fixes and other updates could have been backported to these old versions, without bumping the software version number.

In 2016, we also randomly sampled 2048 IPs which were reporting *dropbear_2014.66*, respectively *OpenSSH_5.3*, to understand what kind of systems were running these services. Based on nmap's OS fingerprinting [105], we speculate that those devices were predominately embedded systems such as routers or firewalls. We did not investigate these systems further. On February 25, 2016 the `Shodan.io` search engine reported 2,792,391 IPs reporting *dropbear_2014.66*. Almost all of those IPs belonged to an IP block owned by Comcast Cable. However, querying the network block owned by Comcast Cable again on May 2, 2016, we only found 83,486 devices listening on port 22. On February 4th, 2020 `shodan.io` reported 364,577 IPs reporting *dropbear_2016.74*, with only a handful (approximately 4,000) belonging to Comcast. Curiously, almost 100,000 of these IPs belong to Charter Communications an American telecommunications company and the second-largest cable operator in the United States (only surpassed by Comcast).

[6] A feature highlighted by the Dropbear author on `https://matt.ucc.asn.au/dropbear/dropbear.html`: "A small memory footprint suitable for memory-constrained environments"

4.7.2 SSH Encryption Schemes

Section 4.4.2 highlighted a high number of unique SSH encryption schemes preferred by at least one SSH server. The number of unique schemes increased from 178 in 2016 to 253 in 2019, an increase of 29.6%. We consider a small amount of diversity to be useful, but a large amount to be dangerous since it brings an increased risk of there being obsolete or insecure options and a higher chance of there being vulnerabilities in the complex code paths needed to support so many options.

On the positive side, the 2019 scan shows that a majority of OpenSSH servers prefer the strong SSH encryption scheme SSH-ChaCha20-Poly1305, and many servers prefer AE schemes or use Encrypt-then-MAC instead of the original choice of Encrypt-and-MAC. Since OpenSSH servers make up a large percentage of the total number of servers, these positive signs are also reflected overall. Furthermore, the top preferred SSH encryption schemes for Dropbear servers use CTR-mode, which, in SSH, is a considerable stronger choice than CBC-mode schemes (cf. Chapter 5).

Focusing on OpenSSH, in the 2016 scan, the SSH encryption scheme preferred on most servers was SSH-AES128-CTR-HMAC-MD5, where AES128-CTR was made the default encryption algorithm choice in OpenSSH version 5.2 (released 23/02/2009) and HMAC-MD5 was made the default MAC algorithm choice in OpenSSH version 1.22[7] (released 05/03/2001 – the first publicly released OpenSSH version). The default preferred MAC algorithm was first changed again in OpenSSH version 6.2 (released 22/03/2013) to the Encrypt-then-MAC algorithm version HMAC-MD5-etm. Nonetheless, the most popular preferred SSH encryption scheme for OpenSSH was still using HMAC-MD5 in our 2016 scan. A 3 year period was not enough for the default algorithm change to propagate into the majority of OpenSSH servers.

OpenSSH version 6.7 (released 06/10/2014) changed the default MAC algorithm again, this time to the algorithm UMAC-64-etm. The algorithm change was done in a period with much internal code refactoring. This might be the reason there is no mention of the algorithm change in the release notes published by the OpenSSH developers. However, the change is visible in the Git commit `9235a030`[8] made on

[7] See `https://github.com/openssh/openssh-portable/commit/ec26fb166788728d7ccafe85730ccf04f3a4885b`. MD5 was made the default choice over SHA1.

[8] `https://github.com/openssh/openssh-portable/commit/9235a030ad1b16903fb495d81544e0f7c7449523`

20/04/2014 (6 months prior to the 6.7 release). In our 2016 scan, the number of servers preferring an SSH encryption scheme that uses the UMAC MAC is small. CHAHCA20-POLY1305 was made the default encryption algorithm as of OpenSSH version 6.9 (released 01/07/2015). Unsurprisingly, this change is also not reflected in the 2016 scan. However, the default encryption and MAC algorithm changes are clearly visible in the 2019 scan, 4-5 years later.

It is clear from the above discussion that any change in a default algorithm (at least for OpenSSH) takes a substantial number of years to propagate to the SSH ecosystem. This is also reflected in the number of different OpenSSH versions still active, some servers using versions first published 20 years ago.

4.7.3 Vulnerable Servers

The number of OpenSSH servers that are vulnerable to an attack from Chapter 5 found in the 2019 scan increased compared to the 2016 scan. This is even true if we only focus on the APW attack. Overall, the number of vulnerable OpenSSH servers has more than doubled in the 2019 scan compared to the 2016 scan.

For Dropbear servers, the number of vulnerable servers has decreased from 816,359 servers in the 2016 scan to 166,234 servers in the 2019 scan. But taking into account that we found markedly less Dropbear servers in the 2019 scan compared to the 2016 scan the situation changes. In 2016, (approximately) 1 in every 12 servers was vulnerable. In 2019, the fraction of vulnerable Dropbear servers had increased to (approximately) 1 in every 6 servers. As with OpenSSH, this is an increase of about 100%.

Chapter 5

Attacks on SSH's CBC-mode

In this chapter, we present new attacks against CBC-mode SSH encryption schemes (henceforth referred to as CBC-mode). We begin by recalling the plaintext recovery attack on CBC-mode from [5]. We then describe a variant of the attack that applies to Dropbear's implementation of CBC-mode. We go on to describe the countermeasure to the attack that was introduced in OpenSSH 5.2.[1] *We proceed to present three new attacks against CBC-mode in OpenSSH. The first attack exploits a bug in the OpenSSH source code. The second attack breaks the countermeasure implemented against [5]. The third attack breaks the countermeasures that were implemented against our second attack. At the end of this chapter, we discuss the impact of our attacks.*

5.1 The Albrecht-Paterson-Watson Attack

We recall the attack from [5] using its text. This attack applied to OpenSSH up to and including version 5.1. In compliance with SSH as described in Section 2.4.2, OpenSSH 5.1 (and earlier) uses CBC-mode with interpacket chaining and random padding by default. OpenSSH 5.1 decrypts the first block of a BPP packet as soon as it is received and extracts the packet length field from the corresponding plaintext block. OpenSSH 5.1 rejects any packets whose packet length field does not satisfy two sanity checks. The first sanity check verifies that the encoded length (measured in bytes) in the packet length field is less than 5 or greater than $256 \cdot 1024 = 2^{18}$ (we refer to this check as the *packet length check*). If this check fails, an SSH error is immediately send by the server to the client and the connection is subsequently terminated. If the check passes, the second sanity check is performed. In this check, the server verifies that the total number of bytes expected in the packet (excluding a

[1] In this chapter, we refer to OpenSSH version X as *OpenSSH X*.

possible MAC field) is a multiple of the block size (we refer to this check as the *block size check*). If this check fails, the connection is immediately terminated *without* sending an SSH error (if one of the two sanity checks fail, a network eavesdropper can therefore derive exactly which of the two checks that failed). If the two sanity checks pass, the server continues to accept data on the connection until sufficient data has arrived; here sufficiency is determined by the content of the packet length field and the size of the MAC field. MAC verification then takes place. If the data has been tampered with, this will fail with high probability, leading to a termination of the connection.

We will use k to denote the key of our block cipher F having inverse F^{-1}. We can assume F to be fixed for the duration of a connection. We let L denote the block size of this block cipher in bytes (so $L = 8$ for `3des` and $L = 16$ for `aes128`). Then CBC-mode in OpenSSH 5.1 operates as follows: given a sequence $p_1, p_2, \ldots p_n$ of plaintext blocks making up a packet, we have:

$$c_i = \mathsf{F}_{\mathsf{k}}(c_{i-1} \oplus p_i), \quad i = 1, 2, \ldots, n,$$

where c_0, the IV, is taken as the last block of the previous SSH BPP ciphertext. Hence

$$p_i = c_{i-1} \oplus \mathsf{F}_{\mathsf{k}}^{-1}(c_i), \quad i = 1, 2, \ldots, n.$$

Assume now that an attacker collects a target ciphertext block c_i^* from an established SSH connection, from some SSH BPP packet. Let c_{i-1}^* denote the ciphertext block preceding the target block, and let p_i^* denote the corresponding target plaintext block. We have $p_i^* = c_{i-1}^* \oplus \mathsf{F}_{\mathsf{k}}^{-1}(c_i^*)$.

The attacker now simply injects the single block c_i^* as the first block of a new packet on the SSH connection. OpenSSH 5.1 will compute as the first block of plaintext for this new packet $p_1' = IV \oplus \mathsf{F}_{\mathsf{k}}^{-1}(c_i^*)$, where IV is the last ciphertext block of the preceding BPP packet.

Combining the two preceding equations, we have:

$$p_i^* = c_{i-1}^* \oplus p_1' \oplus IV. \tag{5.1}$$

We are interested in two cases: the attacker sees either a termination of the TCP connection over which the SSH connection is running without an SSH error message (indicating a failure of the block size check) or the SSH connection enters a state in which it is waiting for more data. In both cases, p_1' must have passed the packet length check. But this only happens if the packet length field in p_1' lies between 5 and 2^{18}, which in turn occurs only if the first 14 bits of p_1' are all zero.[2] From this information and equation (5.1), we can calculate the first 14 bits of p_i^*.

To assess the success probability of this attack, we need only calculate the probability that the packet length check passes. We may assume that c_n, obtained as the last ciphertext block of the preceding BPP packet, acts as a random IV with respect to the block c_i^*. Hence the content of the packet length field in p_1' can be regarded as being a random 32-bit value. Since there are $2^{18}+1$ values for which a 32-bit number is equal or less than 2^{18}, and there are 6 values $(0, 1, 2, 3, 4, 5)$ for which a 32-bit number is equal or less than 5, the packet length check will pass with probability:

$$\frac{2^{18}+1-6}{2^{32}} = \frac{1}{2^{14}}\left(1-\frac{5}{2^{16}}\right) \approx \frac{1}{2^{14}}$$

This attack can be extended to recover 32 bits of plaintext by counting the number of bytes consumed by the SSH server before it terminates the connection because of a MAC failure. In this case, the block size check must also pass. For AES, this means we must have $4 + PL \bmod 16 = 0$, where PL is the numeric value encoded in the packet length field in p_i' and 4 is the size of the packet length field (in bytes). Only one of the 2^4 possible values of the 4 least significant bits of PL make this equation true, namely, the value 12. The other, 14 highest-order bits of the 18 least significant bits, can attain any of the 2^{14} possible values. Hence, the attack's success probability is reduced to:

$$\frac{2^{14}+1-1}{2^{32}} = \frac{1}{2^{18}}.$$

5.1.1 Applying the Albrecht-Paterson-Watson attack to Dropbear

The Dropbear code for CBC-mode (in all versions of Dropbear) performs similar sanity checking to OpenSSH 5.1, but the details are different. Assume a block size of 16 bytes (AES) and a maximum MAC tag length of 32 bytes (corresponding to using HMAC with SHA-256 as the MAC algorithm). From Dropbear's `packet.c`, we have:

```
  len = buf_getint(ses.readbuf) + 4 + macsize;
```

[2] It is also possible that the packet length field is exactly equal to 2^{18}, but this is much less likely given that IV can be treated like a random block.

so that `len` is set to the content of the packet length field plus the MAC output size plus 4. Then:

```
if ((len > RECV_MAX_PACKET_LEN) ||
  (len < MIN_PACKET_LEN + macsize) ||
  ((len - macsize) % blocksize != 0)) {
  dropbear_exit("Integrity error (bad packet size %u)", len);
}
```

so that packet processing terminates immediately if `len` is too large, too small, or does not satisfy the usual block size check. Here, the value of `RECV_MAX_PACKET_LEN` is set to 35,000 and `MIN_PACKET_LEN` is set to 16 by earlier **`#define`** macros. Therefore, if `len` passes the checks, the most significant 16 bits of the packet length field must be 0 (otherwise `len` would be larger than 35'000). To determine the number of possible values that will pass the sanity checks, we first count the number of valid values that are greater or equal to 2^{15}. If the 16th most significant bit is set, then only 137 of the possible values are valid. We next count the number of valid values less than 2^{15}. Of the least significant 15 bits, the 4 lowest-order bits (assuming big-endian) must, again, equal 12, because of the block size check. This gives 2^{11} possible valid values that are less than 2^{15}. In total, there are, therefore, approximately $2^{11} + 137 \approx 2^{11.1}$ possible values out of 2^{32} total values that pass the sanity checks in Dropbear. Therefore, with probability roughly $1/2^{20.9}$ the sanity checks pass, the connection is not terminated, and the attacker learns about 20 bits of plaintext. The attacker can then go on to learn 32 bits of plaintext as in the original attack. The reason that the attack's success probability is lower than for OpenSSH 5.1 is that Dropbear's length test is more stringent by only allowing packets of length at most 35,000 bytes.

5.1.2 OpenSSH Patch against Albrecht-Paterson-Watson Attack

Starting with version 5.2, OpenSSH implements a CBC-mode-specific countermeasure against the attack described above, as follows (from `packet.c`). If the packet length field has the wrong size then a function `ssh_packet_start_discard` is called:

```
        state->packlen =
            PEEK_U32(sshbuf_ptr(state->incoming_packet));
        if (state->packlen < 1 + 4 ||
            state->packlen > PACKET_MAX_SIZE) {
#ifdef PACKET_DEBUG
            fprintf(stderr, "input: \n");
            sshbuf_dump(state->input, stderr);
            fprintf(stderr, "incoming_packet: \n");
            sshbuf_dump(state->incoming_packet, stderr);
#endif
            logit("Bad packet length %u.", state->packlen);
            return ssh_packet_start_discard(ssh, enc, mac,
                state->packlen, PACKET_MAX_SIZE);
        }
```

The same function is called if the block size check fails:

```
  if (need % block_size != 0) {
    logit("padding error: need %d block %d mod %d",
        need, block_size, need % block_size);
    return ssh_packet_start_discard(ssh, enc, mac,
        state->packlen, PACKET_MAX_SIZE - block_size);
  }
```

The `need` variable is equal to the length of the SSH packet length field (4 bytes), the length encoded in the length field excluding the length of the block already decrypted (16 bytes, for AES). Finally, the function is also called if the MAC check, when eventually performed, fails:

```
    if (timingsafe_bcmp(macbuf, sshbuf_ptr(state->input),
        mac->mac_len) != 0) {
      logit("Corrupted MAC on input.");
      if (need > PACKET_MAX_SIZE)
        return SSH_ERR_INTERNAL_ERROR;
      return ssh_packet_start_discard(ssh, enc, mac,
          state->packlen, PACKET_MAX_SIZE - need);
    }
```

Calling the function `ssh_packet_start_discard` causes the server to wait for a certain number `packet_discard` bytes:

```
static int
ssh_packet_start_discard(struct ssh *ssh, struct sshenc *enc,
    struct sshmac *mac, u_int packet_length, u_int discard)
{
  struct session_state *state = ssh->state;
  int r;

  if (enc == NULL || !cipher_is_cbc(enc->cipher)
      || (mac && mac->etm)) {
    if ((r = sshpkt_disconnect(ssh, "Packet corrupt")) != 0)
      return r;
    return SSH_ERR_MAC_INVALID;
  }
  if (packet_length != PACKET_MAX_SIZE && mac && mac->enabled)
    state->packet_discard_mac = mac;
  if (sshbuf_len(state->input) >= discard &&
     (r = ssh_packet_stop_discard(ssh)) != 0)
    return r;
  state->packet_discard = discard - sshbuf_len(state->input);
  return 0;
}
```

Afterwards, once a total of `PACKET_MAX_SIZE` bytes have arrived, `ssh_packet_stop_discard` is called. This function computes a MAC over `PACKET_MAX_SIZE` bytes and then terminates the connection (the MAC algorithm being the one agreed during the initial key establishment):

```
int
ssh_packet_stop_discard(struct ssh *ssh)
{
  struct session_state *state = ssh->state;
  int r;

  if (state->packet_discard_mac) {
    char buf[1024];

    memset(buf, 'a', sizeof(buf));
    while (sshbuf_len(state->incoming_packet) <
        PACKET_MAX_SIZE)
      if ((r = sshbuf_put(state->incoming_packet, buf,
          sizeof(buf))) != 0)
        return r;
    (void) mac_compute(state->packet_discard_mac,
        state->p_read.seqnr,
        sshbuf_ptr(state->incoming_packet), PACKET_MAX_SIZE,
        NULL, 0);
  }
  logit("Finished discarding for %.200s port %d",
      ssh_remote_ipaddr(ssh), ssh_remote_port(ssh));
  return SSH_ERR_MAC_INVALID;
}
```

The overall intention of the countermeasure is to mask the side-channel information used by the attacker in the attack of [5]: now, whenever an error occurs, the implementation waits until `PACKET_MAX_SIZE` bytes have arrived and only then disconnects. This is because of the different settings of the value of `discard` in the different calls to ssh_packet_start_discard.

Note that a MAC over PACKET_MAX_SIZE bytes is eventually computed in ssh_packet_stop_discard in all three cases where ssh_packet_start_discard is called. In two of the three cases, the function is called at the start of packet processing (due to the length check or block size check failing); in the third case it is called only after a MAC verification has already failed.

5.2 Timeline of New Attacks

Below we present three *new* attacks against CBC-mode in OpenSSH. The second attack was the first to be discovered and reported to the OpenSSH developers on 5/5/2016. Attack one and attack three were reported on 1/8/2016. The reason for not presenting the attacks in chronological order is because the vulnerability allowing attack one is a logical bug. Without this bug fixed, attacks two and three, as presented, would not work. It is possible to define variants of the two attacks that would work without the logical bug fixed. However, the presentation of these variants are more convoluted and less self-explanatory. By sacrificing chronologicality, we avoid this.

All code referred to in the next three sections are from the file packet.c in the OpenSSH source code.

5.3 First Attack on OpenSSH CBC-mode

5.3.1 Intuition

This attack exploits a bug in the OpenSSH source code. This bug makes it possible for an adversary to learn whether the packet length / block size check failed or whether the MAC verification failed. This is possible because the latter case will only happen if the server receives strictly more than PACKET_MAX_SIZE bytes. Hence, by sending only PACKET_MAX_SIZE bytes the adversary can observe if the former case happens or not. The adversary can employ a similar attack to the APW attack presented in Section 5.1 with this information.

5.3.2 The Bug

The implementation of the OpenSSH patch for the Albrecht-Paterson-Watson attack contains a logical bug that allows an attacker to mount a similar attack. The bug was present in OpenSSH starting from version 5.2 up until, and including version 7.4.

The bug is due to a subtle error in the computation of the `packet_discard` value. The bug makes it possible to easily distinguish whether the sanity checks on the packet length field, as presented in Section 5.1, failed or not, circumventing the countermeasure.

The computation of `packet_discard` is performed in the function `ssh_packet_start_discard` (shown above) and computed as:

```
  state->packet_discard = discard - sshbuf_len(state->input);
```

where `discard` is the last parameter passed to the function and `state->input` is an internal OpenSSH buffer. This buffer contains encrypted packets and their associated MAC tags, which the server has received but not yet decrypted and verified. When a chunk of bytes from the buffer has been decrypted, it is immediately *consumed* (i.e., deleted) from the buffer via the function `sshbuf_consume`. The function takes as the first input the buffer and as the second input the number of bytes to consume (in FIFO order). In the case of CBC-mode this happens at three locations in the code of which only two are relevant: when the packet length field is retrieved:

```
        if ((r = cipher_crypt(&state->receive_context,
            state->p_send.seqnr, cp, sshbuf_ptr(state->input),
            block_size, 0, 0)) != 0)
            [...]
        if (state->packlen < 1 + 4 ||
            state->packlen > PACKET_MAX_SIZE) {
            [...]
            return ssh_packet_start_discard(ssh, enc, mac,
                state->packlen, PACKET_MAX_SIZE);
        }
        if ((r = sshbuf_consume(state->input, block_size)) != 0)
            goto out;
```

and when the rest of the packet is decrypted:

```
  if ((r = cipher_crypt(&state->receive_context, state->p_read.seqnr,
    cp, sshbuf_ptr(state->input), need, aadlen, authlen)) != 0)
    goto out;
  if ((r = sshbuf_consume(state->input,
    aadlen + need + authlen)) != 0)
    goto out;
```

The third "consume" occurs when the MAC verification has been passed, but this has no relevance to the attack. Since `aadlen` = 0 and `authlen` = 0 when using CBC-mode in OpenSSH, the total number of bytes consumed from the input buffer is `block_size` when the packet length has been retrieved and `block_size` + `need` bytes after the packet has been decrypted.

Now consider the three failure cases where the function `ssh_packet_start_discard` is called. We will compute the number of packet discard bytes in each failure case. Note, no new data is inserted into the input buffer during packet decryption.

Packet length check fails: The value of the `discard` parameter is `PACKET_MAX_SIZE`. The number of packet discard bytes is therefore computed as:

`PACKET_MAX_SIZE` − `sshbuf_len(state->input)`.

Block size check fails: The value of the `discard` parameter is `PACKET_MAX_SIZE` − `block_size`. The number of packet discard bytes is therefore computed as:

`PACKET_MAX_SIZE` − `block_size` − `sshbuf_len(state->input)`.

Since a `block_size` number of bytes have been consumed from the input buffer prior to the block size check, the number of discard bytes computed is the same as when the packet length check fails.

MAC verification check fails: The value of the `discard` parameter is `PACKET_MAX_SIZE`−`need`. The number of packet discard bytes is therefore computed as:

`PACKET_MAX_SIZE` − `need` − `sshbuf_len(state->input)`.

In the case of CBC-mode `need` = 4 + `state->packlen` − `block_size`, where `packlen` is the value encoded in the 4-byte packet length field. Prior to the MAC verification a total of `block_size` + `need` bytes is consumed from the buffer `state->input`. But this means that the number of packet discard bytes computed in this failure case is 16 bytes more than in the two previous failure cases; the initial block of bytes, of length `block_size`, is *mistakenly* counted twice.

Because of the discrepancy, if either the packet length check or block size check fails, the server waits until it has received a total of `PACKET_MAX_SIZE` bytes before returning an error. However, in the case of a MAC verification error the server waits for a total of `PACKET_MAX_SIZE` + `block_size` number of bytes before it returns an error.

5.3.3 Exploiting the Bug

As in [5] (and as explained in Section 5.1), the attacker gathers any target ciphertext block c_i^* from an SSH connection and injects it so that it is interpreted at the server as the first ciphertext block of a new SSH packet. Then the attacker sends a further `PACKET_MAX_SIZE` $- 16$ bytes (for AES and, generally, 16 must be replaced by the block size of the underlying block cipher) to the server for a total of `PACKET_MAX_SIZE` bytes sent. There are two cases:

1. If the server replies with an error and drops the connection, it means that either the packet length check failed or the block size check failed. For a 16-byte block cipher (AES) and assuming the IV is uniformly random, this happens with probability roughly $1 - 2^{-18}$ because `PACKET_MAX_SIZE` is set to 2^{18} in the length check and the block size check is a 4-bit condition.
2. If the server remains silent the two sanity checks must both have passed and the MAC check must have failed, causing the server to wait for an additional `block_size` bytes. This case arises with probability roughly 2^{-18}.

In the latter case, an attacker can learn the first 14 bits and the last 4 bits of the 32 most significant bits of the target plaintext with probability 2^{-18} using the same method as in [5]. Note that the attack cannot be extended to recover the remaining 14 bits, as in the Albrecht-Paterson-Watson attack, because connection termination is no longer dependent on the encoded length in the packet length field.

5.3.4 OpenSSH Patch against Attack One

OpenSSH 7.5[3] removes the double counting of the initial block when computing the number of packet discard bytes after a MAC verification failure:

```
        /* Not EtM: check MAC over cleartext */
        if (!mac->etm && (r = mac_check(mac, state->p_read.seqnr,
            sshbuf_ptr(state->incoming_packet),
            sshbuf_len(state->incoming_packet),
            sshbuf_ptr(state->input), maclen)) != 0) {
            [...]
            return ssh_packet_start_discard(ssh, enc, mac,
                sshbuf_len(state->incoming_packet),
                PACKET_MAX_SIZE - need - block_size);
        }
```

[3] https://github.com/openssh/openssh-portable/commit/0fb1a617a07b8df5de188dd5a0c8bf293d4bfc0e

The change makes the total number of bytes the server waits for the same in each of the three failure cases.

5.4 Second Attack on OpenSSH CBC-mode

5.4.1 Intuition

The second attack essentially replaces the byte-counting side-channel of [5] with a timing side-channel. Through this timing-channel the adversary can learn whether the packet length / block size check failed or whether the MAC verification failed. The latter case would exhibit a slower response time that the former case. This again allows a similar attack to that described in Section 5.3. There are some added complexity compared to the first attack in the sense that the second batch of bytes (PACKET_MAX_SIZE $-$ 16) send must be send quickly to ensure the side-channel signal is strong enough. The data transfer urgency is because we want to measure the MAC processing time and a big data transfer latency would erase that signal.

5.4.2 Description

As before, the attacker gathers any target ciphertext block c_i^* from an SSH connection and injects it so that it is interpreted at the server as the first ciphertext block of a new SSH packet. Then the attacker sends, as quickly as possible, a further PACKET_MAX_SIZE $-$ 16 bytes[4] (for AES) to the server. There are two cases:

1. If either the packet length check or the block size check on the packet length field fails, then when ssh_packet_start_discard is called, the server performs a single MAC computation over PACKET_MAX_SIZE bytes. As before, this case happens with probability $1 - 2^{-18}$ for a block cipher with a 16 byte block size (AES).
2. If both sanity checks pass, then a MAC verification over the number of bytes encoded in the packet length field is carried out. Further, when the MAC verification fails (with overwhelming probability), ssh_packet_start_discard is called. This involves a second MAC computation over PACKET_MAX_SIZE bytes. This case arises with probability roughly 2^{-18}.

[4] If the patch for attack one had not been applied, an attacker must send a total of PACKET_MAX_SIZE + 16 bytes (for AES) to trigger the second MAC computation in the case where the MAC verification fails.

In the second case, an additional MAC computation is performed. Assuming that the attacker can deliver data fast enough to the server that it does not stall while waiting for incoming data to process, then the additional MAC computation will show up as a small delay in the time taken at the attacker to observe an SSH connection termination. The length of the delay is roughly proportional to the amount of data over which the first MAC is computed, which in turn is closely related to the content of the packet length field.

In the basic form of the attack, we assume that the packet length field is randomised, so that with probability 1/2 the size of the packet length field is at least 2^{17} in the second case. (Here we rely on the IV being effectively random.) Hence, the time difference between the first and second cases is the time needed for a MAC computation over at least 2^{17} bytes. For an HMAC-based MAC algorithm with a 64-byte compression function (as in MD5, SHA-1 and SHA-256), this equates to at least 2^{11} compression function evaluations, with each one taking a few hundred clock cycles on a modern CPU. Thus, the time difference is on the order of a few hundred thousand clock cycles, or a few hundred microseconds, which is easily detectable remotely over a network. By comparison, the Lucky 13 attack [6] on SSL/TLS, of which this attack is reminiscent, showed that it is possible to remotely measure timing differences equating to a *single* compression function evaluation, albeit under ideal network conditions. Here the timing signal is at least 2^{11} times as big. On the other hand, this attack assumes that reading 2^{18} bytes from the network as requested by `ssh_packet_start_discard` is sufficiently fast to not drown this timing signal with network jitter.

In summary, with overall probability 2^{-19} (probability 2^{-18} for the second case to occur and probability 1/2 that the size is at least 2^{17}), the attacker sees a measurable delay for SSH connection termination, indicating that the sanity checks have passed. As before, this leaks 18 bits of the plaintext information to the attacker; first 14 bits and the last 4 bits of the 32 most significant bits of the target plaintext. This basic form of the attack is already better than random guessing because it provides confirmation of the unknown plaintext bits. Moreover, assuming the target plaintext is sent in an identifiable ciphertext block across many connections, then the attack can be repeated over multiple connections to increase the success probability. Section 5.7 discusses methods that can optimise this attack.

As stated earlier, attack two, as presented, would not work if the patch against attack one has not been applied. We did not initially uncover attack one because our implementation of attack two, for various SSH encoding reasons, actually sent more than PACKET_MAX_SIZE + 16 bytes to the server, triggering the second MAC computation in the MAC verification failure case.

5.4.3 OpenSSH Patch against Attack Two

The second attack was patched in OpenSSH version 7.3. The mitigation makes the number of bytes passed to HMAC in ssh_packet_stop_discard depend on the number of bytes *already* processed by HMAC prior to entering this function so that the total number of bytes processed by HMAC always adds up to (roughly) PACKET_MAX_SIZE:

```
static int
ssh_packet_start_discard(struct ssh *ssh, struct sshenc *enc,
    struct sshmac *mac, size_t mac_already, u_int discard)
{
  [...]
  /*
   * Record number of bytes over which the mac has already
   * been computed in order to minimize timing attacks.
   */
  if (mac && mac->enabled) {
    state->packet_discard_mac = mac;
    state->packet_discard_mac_already = mac_already;
  }
  if (sshbuf_len(state->input) >= discard)
    return ssh_packet_stop_discard(ssh);
  state->packet_discard = discard - sshbuf_len(state->input);
  return 0;
}
```

This patch is meant to approximately equalise the total time spent on HMAC computations. Below we present an attack that shows this is not true if the attacker sends bytes to the server in a clever way.

5.5 Third Attack on OpenSSH CBC-mode

5.5.1 Intuition

The third attack breaks the countermeasure against the second attack. The attack is a slightly modified version of the second attack. The main difference between the two attacks is how the timing is measured. In the second attack, the case where the MAC verification failed would exhibit a slower response than the case in which the packet length / block size check failed. In this attack, the relationship is reversed e.g. the case where the MAC verification failed exhibits a faster response. This is accomplished by only sending PACKET_MAX_SIZE minus 1 bytes to the server in the second batch (compared to sending PACKET_MAX_SIZE in the second batch in Attack Two). This means that the server would have already computed the first MAC computation in the case where the MAC verification failed and is waiting to compute the second MAC verification over PACKET_MAX_SIZE minus the value encoded in the packet length field bytes.

5.5.2 Description

As before, the attacker gathers any target ciphertext block c_i^* from an SSH connection and injects it so that it is interpreted at the server as the first ciphertext block of a new SSH packet. The first 32 bits of the decryption of the target block will be used to construct the packet length field. Then the attacker sends a further PACKET_MAX_SIZE $- 16 - 1$ bytes (for AES) to the server. The attacker now waits for δ seconds before sending a single byte to the server. There are two cases:

1. If either the packet length check or the block size check on the packet length field fails, then as soon as ssh_packet_start_discard is called, it immediately performs a single MAC computation over PACKET_MAX_SIZE bytes. As before, this happens with probability $1 - 2^{-18}$ for a 16-byte block cipher (AES).
2. If both sanity checks pass, then a MAC verification over the number of bytes encoded in the packet length field is carried out. Further, when the MAC verification fails (with overwhelming probability), ssh_packet_start_discard is called. This involves a second MAC computation over PACKET_MAX_SIZE $- PL$ bytes where PL denotes the length encoded in the packet length field. This case arises with probability roughly 2^{-18}.

In the second case, the second MAC computation is performed only *after* the attacker waits δ seconds and sends the last byte. The only exception is if the packet length field encodes PACKET_MAX_SIZE, which happens with probability 2^{-18} for a 16-byte block size.

The attack takes the same form as attack two but with the timing delay occurring when the MAC verification fails except when either the packet length check of block size check fails. The timing signal is proportional to the encoded length; the larger the encoded length, the more significant the timing signal is. Because of the (low) probability of having an encoding of PACKET_MAX_SIZE the success probability for the attacker is 2^{-19}, using the same arguments used to compute the attack success probability in attack two.

The value δ is lower-bounded by the time it takes for the server to process the first PACKET_MAX_SIZE $- 16 - 1$ bytes (for AES) sent by the attacker. Choosing δ sufficiently big, the attack is always possible to perform. Here we are only interested in knowing that the attack is always possible, which is why have not experimentally analysed the optimal value for δ. But setting δ to 1 second, should leave sufficient time for the server to process the first PACKET_MAX_SIZE $- 16 - 1$ bytes (for AES) sent by the attacker. It might be possible to significantly lower δ in practice.

5.5.3 OpenSSH Patch against Attack Three

The OpenSSH developers have decided not to implement any further countermeasures. We recommended, and still recommend, to revert the patch against attack two. Our main reason for this recommendation is that attack three does not require an attacker to submit bytes quickly. Hence, practically, attack three is easier to carry out than attack two, with the potential to be applicable in a greater number of situations. However, CBC-mode has been disabled in OpenSSH since version 6.7, and the OpenSSH developers do not recommend enabling it.

5.6 Experimental Results

We verified the conditions of all three attacks using Paramiko [47] as the client and sending a different amount of data depending on the attack.

For attack one, we used OpenSSH 7.3 as the server and verified that sending 2^{18} (the default value of PACKET_MAX_SIZE) bytes would not make the server stall if either the packet length check or the block size check fails. But if the sanity checks pass, the server would indeed stall, waiting for a further 16 bytes before proceeding.

We verified the conditions of attack two using OpenSSH version 7.2 as the server, see Table 5.1 for a summary of the results mentioned below. In particular, we verified that under our attack, an OpenSSH server indeed processed 2^{18} or (approximately)

	Length check verification failed	MAC verification failed
Number of MAC computations	1	2
Length of MAC input	2^{18} bytes	$2^{18} + 2^{17}$ bytes
Total time to process MAC(s)	600ms	1200ms
Total time for server to discard packet	880ms	1500ms

Table 5.1: Summary of experiments.

$2^{18} + 2^{17}$ bytes with HMAC if one of the sanity checks failed, or the MAC check failed, respectively. We also performed some basic timing experiments with the following results. If we flipped a bit in the first block of an SSH BPP packet to distort the packet length field, it took about 600 microseconds to compute the MAC on our test system, and we waited for 209 microseconds for additional dummy data to arrive on a loopback device (we started timing this in `ssh_packet_start_discard`). The overall time from sanity checking the packet length field to discarding the connection was about 880 microseconds. If we flipped a bit in a later block so that only the MAC check fails, OpenSSH computed two MACs, which took about 1200 microseconds on our target system. The overall time from sanity checking the length field to discarding the packet was about 1500 microseconds. All timings were done on the server and packets from the client were sent over loopback. Hence, these timings reflect a best case scenario for an attacker. Given that the feasibility of timing side-channels over networks is well established for timing signals much smaller than this magnitude [6], we saw no need to pursue further experiments in a more realistic network environment.

The conditions for attack three were verified using OpenSSH version 7.3 and in a similar way to how the conditions for attack two were verified. In particular, a similar significant timing difference was observed for attack three as was seen in attack two. The magnitude of the difference is dependent on the length encoded in the length field.

5.7 Extensions and Variants of Attack Two and Three

In the case of attacks two and three, if the timing difference can be measured with higher resolution, then more bits of plaintext can be recovered, since, when both sanity checks pass, the additional time taken to verify HMAC on the plaintext data relates closely to the remaining unknown bits of the 32-bit packet length field, namely, the 14 "middle" bits located between the 14 MSBs and the 4 LSBs (in big-endian representation). Specifically, the more significant bits of these middle bits are more easily recovered, while, assuming HMAC with a 64-byte compression function is used, the least 5 significant bits of these middle bits are unlikely to be recoverable; 64 bytes is 2^9 bits which only requires one compression function call, so there would not be anything to time. Assuming the target plaintext is sent in an identifiable ciphertext block across many connections, then the attack can be repeated over multiple connections to more accurately determine the timing difference.

In another variant of the attack, we assume the attacker already knows the most significant 14 bits of the plaintext corresponding to the packet length field in the target ciphertext block. (This makes the attack a "partially known plaintext recovery attack".) This enables the attacker to wait for an IV such that the sanity checks on the packet length field will pass; on average, this will require waiting for (approximately) 2^{14} packets before injecting the target ciphertext block. The attacker can then recover the 4 bits corresponding to the block size check, now with probability 2^{-4}. Again, the attacker can go on to recover more unknown plaintext bits with high probability if the timing differences can be measured with higher resolution. A similar attack is possible if the attacker knows the least significant 4 bits of the plaintext corresponding to the packet length field in the target ciphertext block, involving waiting for 2^4 packets only, but with success probability 2^{-14} in recovering 14 bits of plaintext. Likewise, a variant attack is possible if the attacker knows the most significant 14 bits *and* the least significant 4 bits of the 32-bit field corresponding to the packet length field in the target ciphertext block. In all variants, the attacks can be iterated assuming the target plaintext is sent in an identifiable ciphertext block across many connections.

Variant attacks in which fewer bits of known plaintext are assumed and which have lower success probability (but which also need to wait for fewer packets before injecting the target ciphertext block) should now be self-evident. Note that the "known bits" variants of these attacks are also possible against OpenSSH 5.1, yielding a new variant of the attacks of [5].

5.8 Practical Impact, Mitigations and Recommendations

The attacks presented above succeed with relatively low probability but can be iterated to increase their success rates. They are therefore potentially serious for any applications using SSH that automatically reconnect and retransmit sensitive data on SSH connections. Given the widespread usage of SSH for controlling remote access to high-security systems, we believe the attacks should be mitigated in all SSH implementations.

The simplest mitigation is to stop using CBC-mode encryption in SSH. As our statistics show (cf. Chapter 4), other modes that are immune to this style of attack are widely available. Indeed, from the work of [114], we know that CTR-mode is invulnerable to such attacks. In Chapter 6, we will prove that OpenSSH's SSH encryption schemes SSH-ChaCha20-Poly1305, SSH-Generic-EtM and SSH-AES-GCM are not vulnerable either.

There is currently no patch implemented in OpenSSH (as of version 8.1) against attack three. A best possible mitigation entails the implementation of an Initialise-Update-Finalise (IUF) interface for the MAC operation. The Update operation would then be invoked on plaintext every time a new chunk of ciphertext arrives, and only one Finalise operation would be used at the very end. Note that this might involve the Update function sometimes being called on an empty plaintext buffer, or on a buffer that is so short that no real cryptographic operations are done. However, this mitigation would require significant refactoring of the parts of the OpenSSH code that process incoming packets. Furthermore, the Finalise operation (if invoked using HMAC) triggers 1-2 compression function calls, leaving an observable residual timing difference.

We therefore strongly recommend against the use of CBC-mode in SSH and fully support the decision by the OpenSSH developers to disable CBC-mode by default.

Chapter 6

Concrete Security of SSH Encryption Schemes

In this chapter, we analyse the concrete cryptographic security of various SSH encryption schemes implemented in OpenSSH. We focus on OpenSSH because of its rich variety of schemes and because of its widespread use in practice.

When describing an SSH encryption scheme, we abstract away the specific underlying symmetric encryption scheme and, instead, consider different type of schemes. As a result, the theorems presented in this chapter capture a broader set of SSH encryption schemes than what is, in fact, available in OpenSSH.

6.1 Modelling the OpenSSH Code

The encryption and decryption processes in OpenSSH are mainly performed in functions `ssh_packet_send2_wrapped` and `ssh_packet_read_poll2`, respectively. These present what is essentially a single code-path for the various supported SSH encryption schemes. The code appears to have been developed in a step-by-step fashion as counter-measures to the attack of [5] and extra schemes were added. This development approach has arguably resulted in at least one potentially dangerous error being made, concerning when the MAC is checked in the SSH encryption scheme SSH-Generic-EtM. This is discussed in detail in Section 6.4. One contribution of this thesis is to disentangle the various schemes in the OpenSSH code and to present them in a clean and self-contained way, thereby rendering them amenable to formal analysis.

In our analysis of OpenSSH's implementation of the SSH encryption schemes, we endeavoured to be as faithful as possible to the OpenSSH code. However, in building our pseudo-code models, we had to make a few simplifications and modifications which we now describe. We assume throughout that both compression and extra padding are disabled.[1] In order to model connection tear-downs we introduce a flag CLOSED as part of the decryption state; once it is set, the decryption algorithm will only return the empty string. In our pseudo-code models, we append for every full message returned by the decryption algorithm, a special *end of message* symbol (¶). This is needed in the ciphertext fragmentation model to demarcate message boundaries, but of course does not exist in the real code. Finally, we note that the OpenSSH "error" SSH_ERR_MESSAGE_INCOMPLETE does not result in any error that is visible to a network-based attacker, but only serves to indicate that decryption needs more packet data to be able to fully assemble a packet. We leave out this error from our descriptions.

We next discuss two operations that are common for the cryptographic processing across many types of SSH encryption schemes in OpenSSH, relating to padding and sanity checking. We also highlight some important variables used in these operations. Our pseudo-code for these are embedded into the descriptions of each SSH encryption scheme that appears in the following sections (for example, in Figure 6.6 the padding code appears in lines 4-8 in algorithm ssh-fgEtM-Enc and the sanity checking code appears in lines 11-21 in algorithm ssh-fgEtM-Dec). The padding scheme used is the same for all the supported SSH encryption schemes, despite SSH-ChaCha20-Poly1305 and SSH-AES-GCM not strictly requiring any padding, and SSH-Generic-EtM possibly not needing it, depending on the specific underlying symmetric encryption scheme negotiated for use in the SSH-Generic-EtM construction. Padding is recommended [135, Section 6] (but not strictly required) to be random and, in OpenSSH, is computed using the ChaCha20 stream cipher by default or by other methods depending on configuration. For our purposes, we simply assume the padding to be a uniformly random string of appropriate length. During our work, we discovered two integer overflows in OpenSSH relating to the option of adding extra padding. These could cause interoperability problems and allow padding shorter than 4 bytes to arise, violating the requirements in RFC 4253 [135, Section 6]. We filed a bug report for these issues.[2] and they have subsequently been fixed. The sanity checks performed on the length field when an SSH BPP packet is received are the same for SSH-ChaCha20-Poly1305, SSH-Generic-EtM and SSH-AES-GCM. These

[1] An option to allow adding extra padding (implicitly capped at 255 bytes) exists in OpenSSH. This is exclusively used for padding user passwords during authentication and is not included in our descriptions.

[2] https://bugzilla.mindrot.org/show_bug.cgi?id=2566

consist of extracting the value encoded in the length field and verifying that it is in the range $[5, 2^{18}]$ and that it is an integer multiple of the block size. In the descriptions that follows, variable ℓ_{packet} denotes the combined length (in bytes) of the padding length field, payload field and padding, while bsize denotes the block size (in bytes) of the negotiated symmetric encryption scheme. If there is no well-defined block size number, SSH generally defaults to a block size of size 8 (cf. `cipher.c` file in the OpenSSH source code for a list of block sizes). The variable ℓ_{packet} is extracted from the packet length field when a packet is received and denotes the packet length (in bytes). For SSH-ChaCha20-Poly1305 the packet length cannot be extracted directly from the SSH BPP packet, because the packet length field is encrypted, cf. Section 6.3. In our pseudo-code, frag represents the buffer that stores the incoming packet fragments until a whole packet has been received and can be further processed.

In the following, when referring to a packet, we mean the SSH BPP packet consisting of the length field, packet length field, payload and padding field. When referring to a packet, we might mean a packet with or without its associated authentication tag. The terminology *packet fragment* is defined similarly to the terminology *ciphertext fragment* in the ciphertext fragmentation model.

6.2 Adversary Resources Considered

For any adversary in this chapter, we consider the following resources, or a subset of them, depending on the adversary:

- The computation time of the adversary.
- The number of queries made by the adversary.
- The total length of the queries made by the adversary counted in bytes.

We explain the last resource in a bit more detail. If an adversary makes q queries totalling μ bytes, μ is equal to the sum of the length (counted in bytes) of all q queries. To distil further, if the adversary makes queries of the form $(v_1, v_2, \ldots)$, then μ is equal to the sum $|v_1|_{\mathsf{B}} + |v_2|_{\mathsf{B}} + \cdots$ over all such queries.

6.3 SSH-ChaCha20-Poly1305 in OpenSSH

ChaCha20 is the stream cipher defined in [28] and Poly1305MAC is the one-time MAC defined in [113]. We define the following interfaces for ChaCha20 and Poly1305MAC: ChaCha20 takes a 32-byte key k, a variable-length plaintext m, an 8-byte nonce nonce and an 8-byte initial block counter block_ctr and outputs an encryption c of m. Bernstein [28] originally defined ChaCha20 to use a 12-byte nonce and an 4-byte block counter but this was later modified in various documents specifying the combination of ChaCha20 and Poly1305MAC (see below). We write $c \leftarrow \mathsf{ChaCha20}_{\mathsf{k}}(m, \mathsf{nonce}, \mathsf{block\_ctr})$. Reversing the roles of m and c yields the corresponding decryption process. Internally, ChaCha20 makes use of a fixed-output-length pseudo-random function, the ChaCha20 block function ChaCha20-block, a fact which we use in our proofs. The output length of ChaCha20-block is 64 bytes long, and takes as input the key, nonce and block counter. Poly1305MAC takes a 32-byte key k and a variable-length string str and outputs a 16-byte tag τ. We write $\tau \leftarrow \mathsf{Poly1305MAC}_{\mathsf{k}}(\mathsf{str})$.

The generic composition of ChaCha20 and Poly1305MAC is described in RFC 7539 [113] and adapted to SSH in the RFC draft [111] (inspired by the TLS equivalent [99])) which defines the SSH encryption scheme SSH-ChaCha20-Poly1305. In OpenSSH, SSH-ChaCha20-Poly1305 is denoted `chachapoly1305@openssh.com`. The scheme produces a 16-byte MAC tag and encrypts in 64-byte blocks. The nonce used by ChaCha20 consists of a 4-byte sequence number stored as an 8-byte type. The key used is 64 bytes.

When SSH-ChaCha20-Poly1305 is negotiated, the cryptographic processing of a packet deviates from the process described in Section 2.4.2. First, the length field is encrypted using ChaCha20, using the first 32 bytes of the key and the initial block counter set to zero. Second, the remaining part of the packet is encrypted using the last 32 bytes of the key and the initial block counter set to one. That is, two distinct instances of the ChaCha20 algorithm are used to encrypt the two parts. In both instances, the nonce is the SSH packet sequence number. Third, a MAC tag is computed over the entire encrypted packet. The key used here is obtained from a call to ChaCha20 keyed with the last 32 bytes of the key, the 4-byte sequence number (cast to an 8-byte type) as nonce and an initial block counter value of 0, and an all-zero 32-byte plaintext. Note that the sequence number is not in the MAC scope (formally violating Section 6.4

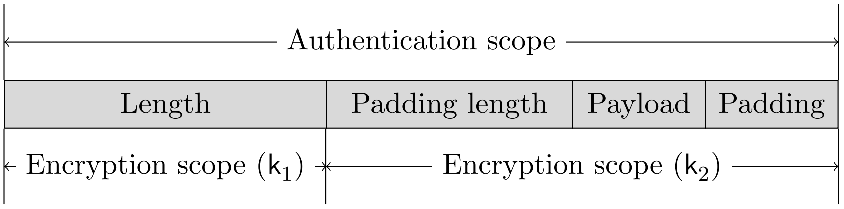
Authentication scope
Length
Padding length
Payload
Padding
Encryption scope (k_1)
Encryption scope (k_2)

Figure 6.1: Cryptographic processing of an SSH BPP packet using the SSH encryption scheme SSH-ChaCha20-Poly1305. k_1 and k_2 represent the two 32-byte keys extracted from the 64-byte key. The size ratios between the boxes do not represent the size ratios between fields in a SSH BPP packet in general.

in [135]) but is integrity protected implicitly through its role in deriving the MAC key. See Figure 6.1 for an illustration of the encryption and authentication scope.

Decryption supports ciphertext fragmentation by first decrypting the length field and checking that it satisfies the usual length requirements. Successive packet fragments are then accumulated until the received MAC tag can be verified against the packet. If the MAC tag is valid, the remaining portion of the packet is decrypted and the padding removed. A detailed description of SSH-ChaCha20-Poly1305 can be found in Figure 6.5 (at the end of this chapter). SSH-ChaCha20-Poly1305 is a symmetric encryption scheme supporting ciphertext fragmentation with associated key space $\mathcal{K} = \mathsf{B}^{64}$, plaintext space $\mathcal{M} = \mathsf{B}^*$, ciphertext space $\mathcal{C} = \mathsf{B}^*$ and error set $\mathsf{S}_\perp = \{\perp_{\mathsf{SSH\_ERR\_CONN\_CORRUPT}}, \perp_{\mathsf{SSH\_ERR\_MAC\_INVALID}}\}$.

We are now ready to state and prove our theorems regarding the concrete security of the SSH encryption scheme SSH-ChaCha20-Poly1305. Because the nonce used is equal to the sequence number, we must impose the following restriction:

$\mathsf{R}_{\mathsf{chachapoly}}$: The adversary must make no more than 2^{32} encryption and decryption queries.

Theorem 4 (SSH-ChaCha20-Poly1305 is INT-sfCTF secure)**.**
Let SSH-ChaCha20-Poly1305 *be the SSH encryption scheme described in Figure 6.5. Then for any* INT-sfCTF *adversary* $\mathcal{A}_{\mathsf{ctf}}$*, respecting restriction* $\mathsf{R}_{\mathsf{chachapoly}}$*, against* SSH-ChaCha20-Poly1305*, there exists a PRF adversary* $\mathcal{A}_{\mathsf{prf}}$ *against the* ChaCha20 *block function* ChaCha20-block *such that:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}ctf}}_{\mathsf{SSH\text{-}ChaCha20\text{-}Poly1305}}(\mathcal{A}_{\mathsf{ctf}}) \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{ChaCha20\text{-}block}}(\mathcal{A}_{\mathsf{prf}}) + \frac{1}{2^{88}}, \qquad (6.1)$$

where $\mathcal{A}_{\mathsf{prf}}$ *runs in time similar to* $\mathcal{A}_{\mathsf{ctf}}$*. If* $\mathcal{A}_{\mathsf{ctf}}$ *makes* q_e *encryption queries totalling*

μ_e bytes and q_d decryption queries totalling μ_d bytes, then $\mathcal{A}_{\mathsf{prf}}$ makes at most $\lceil \frac{\mu_e}{64} \rceil + 2 \cdot q_e + \lceil \frac{\mu_d}{64} \rceil + 2 \cdot q_d$ PRF queries.

Proof of Theorem 4. We prove the theorem through a sequence of games. For each game below, let FORGE represent the event that the adversary wins according to the win condition in Definition 21.

G0 This is the INT-sfCTF game instantiated with SSH-ChaCha20-Poly1305. Hence:

$$\Pr\left[\mathsf{INI},\ \mathcal{A}_{\mathsf{ctf}}^{\mathsf{ENC}(\cdot),\mathsf{DEC}(\cdot)}\ :\ \mathsf{FORGE}\right] = \Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}]\,. \tag{6.2}$$

G1 In the encryption algorithm ssh-ChaCha20-Poly1305-Enc, we replace calls of the form $\mathsf{ChaCha20}_{\mathsf{k}_2}(m, \mathsf{nonce}, \mathsf{block\_ctr})$ with:

$$m[0 : |m|_{\mathsf{B}} - 1]_{\mathsf{B}} \oplus \mathsf{S}[0 : |m|_{\mathsf{B}} - 1]_{\mathsf{B}},$$

where m is either $(\mathtt{0x00})^{32}$ or packet. Similarly, in the decryption algorithm ssh-ChaCha20-Poly1305-Dec, we replace calls of the form $\mathsf{ChaCha20}_{\mathsf{k}_2}(c, \mathsf{nonce}, \mathsf{block\_ctr})$ with:

$$c[0 : |c|_{\mathsf{B}} - 1]_{\mathsf{B}} \oplus \mathsf{S}[0 : |c|_{\mathsf{B}} - 1]_{\mathsf{B}},$$

where c is either $(\mathtt{0x00})^{32}$ or $\mathsf{frag}[4 : 4 + \ell_{\mathsf{packet}}]$. In both cases, S is defined as the string (using the current local nonce and block counter):

$$\begin{aligned} F(\mathsf{nonce}, \mathsf{block\_ctr}) \,\|\, & F(\mathsf{nonce}, \mathsf{block\_ctr} + 1) \\ \|\, & \cdots \\ \|\, & F(\mathsf{nonce}, \mathsf{block\_ctr} + \lfloor \mathsf{len}/64 \rfloor), \end{aligned}$$

where F is a random function with output size 64 bytes and len is equal to the size of m or c (counted in bytes). For any INT-sfCTF adversary $\mathcal{A}_{\mathsf{ctf}}$, we can build a PRF adversary $\mathcal{A}_{\mathsf{prf}}$ against ChaCha20-block. $\mathcal{A}_{\mathsf{prf}}$ runs $\mathcal{A}_{\mathsf{ctf}}$ and simulates for it Game G0 using its own oracle to compute the calls to ChaCha20 under k_2. $\mathcal{A}_{\mathsf{prf}}$ uses bookkeeping to avoid making queries to the PRF oracle with the same input. $\mathcal{A}_{\mathsf{prf}}$ outputs 1 if FORGE occurs and outputs 0 otherwise. Clearly when $\mathcal{A}_{\mathsf{prf}}$'s oracle is instantiated with ChaCha20-block it perfectly simulates Game G0 and when its oracle is instantiated with a random function it provides $\mathcal{A}_{\mathsf{ctf}}$ with a perfect simulation of Game G1. Thus:

$$\Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}] - \Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{ChaCha20\text{-}block}}(\mathcal{A}_{\mathsf{prf}}). \tag{6.3}$$

G2 In this game, we modify how to compute and verify the MAC tag. First, we make the definition of Poly1305MAC more explicit:

$$\mathsf{Poly1305MAC}_{\mathsf{k}_{\mathsf{poly}}}(m) = \langle \mathsf{Poly1305}(r, m) + s \bmod 2^{128} \rangle_{128}, \qquad (6.4)$$

where $\mathsf{k}_{\mathsf{poly}} = (r, s)$, where r and s are both of length 16 bytes. The function Poly1305 on the right-hand side is defined according to Section 2.5.1 in RFC 7539 [113] except we make the following two operations explicit: the operation `"a += s"` corresponding to the addition of s and the operation `"return num_to_16_le_bytes(a)"` corresponding to the modulus operation with 2^{128} (which simply zeroise all bits above the 128 least significant bit). Note, the clamping of r is performed in the function Poly1305. Since s is a 128-bit string and is chosen uniformly at random (it is the output of the random function F in Game G2), the output from Poly1305MAC is uniformly distributed on $\{0,1\}^{128}$ and independent of previous outputs because the nonce is non-repeating (here we use the that $\mathsf{R}_{\mathsf{chachapoly}}$ is respected).

We now make a few modifications to the encryption and decryption oracles.

In the encryption oracle, we do not generate a key for Poly1305MAC. Instead, we sample a random byte-string τ' of length 16 bytes and use τ' as the MAC tag. In addition, we record the tuple $(\mathsf{nonce}', \tau', c'_{\mathsf{length}} \parallel c'_{\mathsf{packet}})$ in a list $\mathcal{L}$. The first component is the nonce, the second component is the sampled MAC tag τ', and the third component is the concatenation of the output from encrypting ℓ_{packet} and packet. We denote these with an apostrophe to distinguish them from the values consumed during decryption.

During decryption, we proceed in one of two possible ways:

(1) If nonce is not a first component of any element in the list $\mathcal{L}$, no tag has been generated by the encryption oracle for that nonce. We then generate a key $\mathsf{k}_{\mathsf{poly}} = (r, s)$ and verify the tag as in Game G1.

(2) If the nonce is a first component of an element in the list $\mathcal{L}$, we can find the corresponding tag τ' that has been computed over $c'_{\mathsf{length}} \parallel c'_{\mathsf{packet}}$. Compute ℓ_{packet} and τ as in Game G1. The tag τ extracted from the assembled packet is verified by first generating r and then checking whether:

$$\begin{aligned} &\mathsf{Poly1305}_r(\mathsf{frag}[0 : 4 + \ell_{\mathsf{packet}}]) \\ &\qquad = \mathsf{Poly1305}_r(c'_{\mathsf{length}} \parallel c'_{\mathsf{packet}}) + \langle \tau \rangle^{-1} - \langle \tau' \rangle^{-1} \bmod 2^{128}, \end{aligned}$$

Note that through these changes, we are merely changing the way we sample

$\mathsf{k_{poly}}$. Specifically, we generate s implicitly by generating τ' instead, and we defer the generation of the r component to the decryption stage. Since s is identically distributed in both Game G1 and Game G2 we have:

$$\Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{ctf}}) : \mathsf{FORGE}] = \Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{ctf}}) : \mathsf{FORGE}]. \tag{6.5}$$

It now remains to bound $\Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{ctf}}) : \mathsf{FORGE}]$. The event FORGE requires the decryption algorithm to output plaintext and this output cannot come from decrypting in-sync packets because it will be filtered out by the decryption oracle. It is a necessary condition for the decryption algorithm to output plaintext that the MAC tag verifies correctly. Since the CLOSED flag will be set as soon as one invalid packet is detected, it suffices to consider only the first out-of-sync packet (note this packet might be received over several queries). It follows that $\Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{ctf}}) : \mathsf{FORGE}]$ is bounded above by the probability of the MAC tag being valid for the first out-of-sync packet.
There are two cases to consider, if no tag was computed by the encryption oracle for the given nonce then irrespective of the packet this probability is equal to 2^{-128}. Otherwise, it is bounded by the almost-Δ-universality of Poly1305 which was proven in [26]. This states that for any pair of differing strings (v, v') and any 16-byte string z, the probability that $\mathsf{Poly1305}(r, v) = \mathsf{Poly1305}(r,)v') + \langle z \rangle^{-1} \bmod 2^{128}$ is bounded by $8\lceil \max(|v|_{\mathsf{B}}, |v'|_{\mathsf{B}})/16 \rceil / 2^{106}$. Let $\mathsf{frag}[0 : 4 + \ell_{\mathsf{packet}}]$ be the first out-of-sync packet. Since a packet has already been output by the encryption oracle for the given nonce $\mathsf{frag}[0 : 4 + \ell_{\mathsf{packet}}]$ and $c'_{\mathsf{length}} \parallel c'_{\mathsf{packet}}$ must be different. We can therefore apply the bound. The maximal packet length permitted by OpenSSH when using SSH-ChaCha20-Poly1305 is $2^{18} + 4 + 16$ bytes, where the terms are the maximal payload length (including the padding length field), the length of the packet length field and the length of the MAC tag, respectively. Setting $\max(v, v')$ to the maximal packet length yields:

$$\Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{ctf}}) : \mathsf{FORGE}] \leq \frac{1}{2^{88}}. \tag{6.6}$$

Note, the maximal payload length is not a hard upper bound in OpenSSH (or in the SSH specification) and can be changed at compile time by modifying the source code. Increasing the maximal payload length by a factor of 2^l gives a further security loss of l bits.

Combining (6.2), (6.3), (6.5) and (6.6) yields the advantage bound (6.1).

Suppose $\mathcal{A}_{\mathsf{ctf}}$ makes q_e encryption queries totalling μ_e bytes and q_d decryption queries totalling μ_d bytes. Then $\mathcal{A}_{\mathsf{prf}}$ makes at most $\lceil \frac{\mu_e}{64} \rceil + 2 \cdot q_e$ PRF queries in all encryption queries. The first term covers the total number of 64-byte blocks of plaintext. The second term covers the total number of blocks of plaintext that are less than 64 bytes long (there can be at most one of these per query because the encoding adds at most 12 bytes) and the query to generate $\mathsf{k}_{\mathsf{poly}}$. A more formal way to see this is first to write: $\mu_e = \sum_i \mu_i$, where each μ_i is the number of bytes queried in the ith encryption query. The number of PRF queries in each encryption query is $\lfloor \frac{\mu_i}{64} \rfloor + 2$ because the encoding adds at most 12 bytes. Then:

$$\sum_{i=1}^{q_e} \left(\left\lfloor \frac{\mu_i}{64} \right\rfloor + 2 \right) \leq \left\lceil \frac{\sum_{i=1}^{q_e} \mu_i}{64} \right\rceil + 2 \cdot q_e = \left\lceil \frac{\mu_e}{64} \right\rceil + 2 \cdot q_e.$$

Similarly, $\mathcal{A}_{\mathsf{prf}}$ makes at most $\lceil \frac{\mu_d}{64} \rceil + 2 \cdot q_d$ PRF queries in all decryption oracle queries. It is clear that $\mathcal{A}_{\mathsf{prf}}$ runs in time similar to $\mathcal{A}_{\mathsf{ctf}}$. □

In Game $\mathsf{G1}$, we not only change the way $\mathsf{k}_{\mathsf{poly}}$ is generated by also how the plaintext is encrypted (and subsequently decrypted). A careful analysis of the proof, will show that the latter step is, in fact, not strictly necessary. The reason for including this in the proof is to help the streamlined nature of the proofs appearing in this section. Excluding the latter step would improve the tightness of the bound (6.1). Specifically, the number of PRF queries would decrease from $\lceil \frac{\mu_e}{64} \rceil + 2 \cdot q_e + \lceil \frac{\mu_d}{64} \rceil + 2 \cdot q_d$ queries to $q_e + q_d$ queries.

Theorem 5 (SSH-ChaCha20-Poly1305 is IND-sfCFA secure).
Let SSH-ChaCha20-Poly1305 *be the OpenSSH encryption scheme described in Figure 6.5. Then for any* IND-sfCFA *adversary* $\mathcal{A}_{\mathsf{sfcfa}}$*, respecting restriction* $\mathsf{R}_{\mathsf{chachapoly}}$*, against* SSH-ChaCha20-Poly1305*, there exists a PRF adversary* $\mathcal{A}_{\mathsf{prf}}$ *against the* ChaCha20 *block function* ChaCha20-block *such that:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}sfcfa}}_{\mathsf{SSH\text{-}ChaCha20\text{-}Poly1305}}(\mathcal{A}_{\mathsf{sfcfa}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{ChaCha20\text{-}block}}(\mathcal{A}_{\mathsf{prf}}) + \frac{1}{2^{87}}, \tag{6.7}$$

where $\mathcal{A}_{\mathsf{prf}}$ *runs in time similar to* $\mathcal{A}_{\mathsf{sfcfa}}$*. If* $\mathcal{A}_{\mathsf{sfcfa}}$ *makes* q_e *encryption queries totalling* μ_e *bytes and* q_d *decryption queries totalling* μ_d *bytes, then* $\mathcal{A}_{\mathsf{prf}}$ *makes at most* $\lceil \frac{\mu_e}{2 \cdot 64} \rceil + 2 \cdot q_e + \lceil \frac{\mu_d}{64} \rceil + 2 \cdot q_d$ *PRF queries.*

Proof of Theorem 5. We prove the theorem through a sequence of games. For each of these games let WIN represent the event that the adversary guesses the bit b correctly.

G0 This is the IND-sfCFA game instantiated with SSH-ChaCha20-Poly1305. Hence:

$$\Pr\left[\mathsf{INI} \,:\, \mathcal{A}_{\mathsf{sfcfa}}^{\mathsf{LR}(b,\cdot,\cdot),\mathsf{DEC}(\cdot)} = b\right] = \Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}]\,. \tag{6.8}$$

G1 Make the same modifications that were made to define Game G1 in the proof of Theorem 4. Also, define $\mathcal{A}_{\mathsf{prf}}$ similarly, except $\mathcal{A}_{\mathsf{prf}}$ outputs 1 if WIN occurs and 0 otherwise. Using the same arguments, we have:

$$\Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] - \Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{ChaCha20\text{-}block}}(\mathcal{A}_{\mathsf{prf}}). \tag{6.9}$$

G2 Make the same modifications that were made to define Game G2 in the proof of Theorem 4. Using the same arguments, we have:

$$\Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] = \Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}]\,. \tag{6.10}$$

G3 In this game, we further modify ssh-ChaCha20-Poly1305-Dec. Let C be the first out-of-sync packet and i the corresponding value of the sequence number (i.e. nonce). We can find t such that $C = \mathsf{S_F}[t : t + 4 + \ell_{\mathsf{packet}} + 16 - 1]_{\mathsf{B}}$ and $C \neq \mathcal{L}_{\mathsf{C}}[i]_{\mathsf{B}}$. It is possible to discover when we are processing C using $\mathsf{S_F}$ and $\mathcal{L}_{\mathsf{C}}$, and thereby identifying C in the string $\mathsf{S_F}$. Let B denote the boolean value that is true if C is being processed and false otherwise.

Replace the MAC checks in Game G2 in ssh-ChaCha20-Poly1305-Dec with the boolean value B and branch if the value is true. In addition, remove the decryption of $\mathsf{S_F}[4 : 4 + \ell_{\mathsf{packet}}]$, the sanity check of the padding length and the assignment to the output buffer msg. Instead, the content of msg can be derived using $\mathsf{S_F}$, $\mathcal{L}_{\mathsf{C}}$ and $\mathcal{L}_{\mathsf{M}}$ (but only as long as B is false). Note, the random function F is no longer used in the decryption oracle.

Let **BAD** denote the event that B is true, and the relevant MAC check in Game G2 did not fail for C. If **BAD** does not occur, Game G2 and Game G3 are identical. Hence:

$$\Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] - \Pr[\mathsf{G3}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] \leq \Pr[\mathbf{BAD}]\,. \tag{6.11}$$

If the event **BAD** occurs in Game G2 the decryption oracle will output an element from the set $\{0, 1, \P\}^*$. This is exactly the event FORGE. Therefore:

$$\Pr[\mathbf{BAD}] \leq \Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{FORGE}]\,. \tag{6.12}$$

A similar analysis to that in the proof of Theorem 4 yields:

$$\Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{FORGE}] \leq \frac{1}{2^{88}}. \tag{6.13}$$

G4 In this game, we set c_{packet} to $\mathsf{S}[0 : |m|_{\mathsf{B}} - 1]_{\mathsf{B}}$ instead of $m \oplus \mathsf{S}[0 : |m|_{\mathsf{B}} - 1]_{\mathsf{B}}$ in the encryption oracle. Recall that the string S was introduced in Game G1 and defined as in the proof of Theorem 4. F is a random function and called on distinct nonces. Therefore, this does not change the distribution of the c_{packet} part returned by the oracle LR. Because the decryption oracle in G3 does not depend on F, we have:

$$\Pr[\mathsf{G3}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] = \Pr[\mathsf{G4}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}]\,. \tag{6.14}$$

The c_{length} part of the output from the encryption oracle, does not depend on the bit b (the input plaintexts have the same length and ChaCha20 is length-regular), and since the two other parts of the output are uniformly and independent random strings that do not depend on b either, we have:

$$\Pr[\mathsf{G4}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] = \frac{1}{2}. \tag{6.15}$$

Combining (6.8), (6.9), (6.10), (6.11), (6.12), (6.13), (6.14) and (6.15) yields the advantage bound (6.7).

Suppose $\mathcal{A}_{\mathsf{ctf}}$ makes e_q encryption queries totalling μ_e bytes and q_d decryption queries totalling μ_d bytes. Then $\mathcal{A}_{\mathsf{prf}}$ makes at most $\lceil \frac{\mu_e}{2\cdot 64} \rceil + 2 \cdot q_e + \lceil \frac{\mu_d}{64} \rceil + 2 \cdot q_d$ PRF queries. The computation is equal to the computation in the proof of Theorem 4, except that the first term can be reduced by a factor $\frac{1}{2}$ because two equal length inputs are queried to the encryption oracle but only one is actually used.

It is clear that $\mathcal{A}_{\mathsf{prf}}$ runs in time similar to $\mathcal{A}_{\mathsf{sfcfa}}$. □

Theorem 6 (SSH-ChaCha20-Poly1305 is BH-CPA secure)**.**
Let SSH-ChaCha20-Poly1305 *be the OpenSSH encryption scheme described in Figure 6.5. Then for any* BH-CPA *adversary* $\mathcal{A}_{\mathsf{bhcpa}}$*, respecting restriction* $\mathsf{R}_{\mathsf{chachapoly}}$*, against* SSH-ChaCha20-Poly1305*, there exist two PRF adversaries* $\mathcal{A}_{\mathsf{prf}}$ *and* $\mathcal{A}'_{\mathsf{prf}}$ *against the* ChaCha20 *block function* ChaCha20-block *such that:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{bh\text{-}cpa}}_{\mathsf{SSH\text{-}ChaCha20\text{-}Poly1305}}(\mathcal{A}_{\mathsf{bhcpa}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{ChaCha20\text{-}block}}(\mathcal{A}_{\mathsf{prf}}) \\ + 2 \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{ChaCha20\text{-}block}}(\mathcal{A}'_{\mathsf{prf}}),\end{aligned} \tag{6.16}$$

where $\mathcal{A}_{\mathsf{prf}}$ *and* $\mathcal{A}'_{\mathsf{prf}}$ *runs in time similar to* $\mathcal{A}_{\mathsf{bhcpa}}$*. If* $\mathcal{A}_{\mathsf{bhcpa}}$ *makes* q_e *queries totalling* μ_e *bytes, then* $\mathcal{A}_{\mathsf{prf}}$ *and* $\mathcal{A}'_{\mathsf{prf}}$ *each make at most* $\lceil \frac{\mu_e}{64} \rceil + 4 \cdot q_e$ *PRF queries.*

Proof of Theorem 6. We prove the theorem through a sequence of games. For each of these games let WIN represent the event that the adversary guesses the bit b correctly.

G0 This is the BH-CPA game instantiated with SSH-ChaCha20-Poly1305. Hence:

$$\Pr\left[\mathsf{INI} \,:\, \mathcal{A}_{\mathsf{bhcpa}}^{\mathsf{LR\text{-}BH}(b,\cdot,\cdot)} = b\right] = \Pr\left[\mathsf{G0}(\mathcal{A}_{\mathsf{bhcpa}}) \,:\, \mathsf{WIN}\right]. \tag{6.17}$$

G1 Make the same modifications that were made to the encryption algorithm in Game G1 in the proof of Theorem 4. Let F be the random function sampled. Also, define $\mathcal{A}_{\mathsf{prf}}$ similarly, except $\mathcal{A}_{\mathsf{prf}}$ outputs 1 if WIN occurs and 0 otherwise. Using the same arguments, we have:

$$\Pr\left[\mathsf{G0}(\mathcal{A}_{\mathsf{bhcpa}}) \,:\, \mathsf{WIN}\right] - \Pr\left[\mathsf{G1}(\mathcal{A}_{\mathsf{bhcpa}}) \,:\, \mathsf{WIN}\right] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{ChaCha20\text{-}block}}(\mathcal{A}_{\mathsf{prf}}). \tag{6.18}$$

G2 In this game, we replace the ChaCha20 calls that use the key k_1, in the same fashion as in Game G1. The random function used is a new function chosen independently from the function F sampled in Game G1. Define the adversary $\mathcal{A}'_{\mathsf{prf}}$ similar to $\mathcal{A}_{\mathsf{prf}}$. Using the same arguments as above we have:

$$\Pr\left[\mathsf{G1}(\mathcal{A}_{\mathsf{bhcpa}}) \,:\, \mathsf{WIN}\right] - \Pr\left[\mathsf{G2}(\mathcal{A}_{\mathsf{bhcpa}}) \,:\, \mathsf{WIN}\right] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{ChaCha20\text{-}block}}(\mathcal{A}'_{\mathsf{prf}}). \tag{6.19}$$

G2 In this game, we do not generate a key for Poly1305MAC. Instead, we generate the tag by sampling a random byte-string τ of length 16 bytes. Using the same argument as in the proof of Theorem 4, it is clear that:

$$\Pr\left[\mathsf{G2}(\mathcal{A}_{\mathsf{bhcpa}}) \,:\, \mathsf{WIN}\right] = \Pr\left[\mathsf{G3}(\mathcal{A}_{\mathsf{bhcpa}}) \,:\, \mathsf{WIN}\right]. \tag{6.20}$$

Since a new nonce is used in each encryption query, and c_{packet} and c_{length} are generated using two independent random functions, the output from the encryption oracle is independent of the bit b in Game G3. Hence:

$$\Pr\left[\mathsf{G3}(\mathcal{A}_{\mathsf{bhcpa}}) \,:\, \mathsf{WIN}\right] = \frac{1}{2}. \tag{6.21}$$

Combining (6.17), (6.18), (6.19), (6.20) and (6.21) yields the advantage bound (6.16).

Suppose $\mathcal{A}_{\mathsf{bhcpa}}$ makes q_e encryption queries totalling μ_e bytes then for each call to the encryption algorithm, $\mathcal{A}_{\mathsf{prf}}$ and $\mathcal{A}'_{\mathsf{prf}}$ will each make at most $\left\lceil \frac{\mu_e}{64} \right\rceil + 4 \cdot q_e$ PRF queries, derived using similar arguments to the ones used in the proof of Theorem 4. The factor four in the last term comes from the fact that both plaintext inputs to the encryption oracles are encrypted.

$\mathcal{A}_{\mathsf{prf}}$ and $\mathcal{A}'_{\mathsf{prf}}$ both run in time similar to $\mathcal{A}_{\mathsf{bhcpa}}$. □

6.4 SSH-Generic-EtM in OpenSSH

This SSH encryption scheme is described briefly in the PROTOCOL file[3] in the OpenSSH codebase but does not seem to be formally documented in the form of an RFC or other appropriate standards bodies. We have therefore extracted our description of it directly from the OpenSSH source code.

The generic Encrypt-then-MAC construction (SSH-Generic-EtM) in OpenSSH allows any combination of supported encryption and MAC algorithms to be run in Encrypt-then-MAC mode. However, we note that the actual implementation of SSH-Generic-EtM in OpenSSH prior to version 7.3 does not implement Encrypt-then-MAC in the expected way. While the MAC tag of a received packet is computed before decryption commences, it is only compared to the received MAC tag after decryption is complete, see Figure 6.2. Presumably, this resulted from implementing SSH-Generic-EtM on top of the implicit Encrypt-and-MAC structuring of the OpenSSH legacy code. As a consequence, the decryption function could produce a plaintext-dependent error before the MAC is checked, opening the code up to attacks involving packet manipulation. For example, suppose CBC-mode using PKCS#7 [92] padding were to be at some point added to the roster of available encryption algorithms in OpenSSH. Then the late MAC check would enable a padding oracle style attack to be mounted. We stress, however, that at this point no attack is known exploiting the late MAC check. This issue was fixed in OpenSSH version 7.3 (cf. Section 1.2).

It would be possible to prove SSH-Generic-EtM secure in our model assuming that any errors thrown by the decryption algorithm can be simulated without knowledge of the secret key (formally, by assuming the existence of an efficient keyless decryption error simulator). But such an approach would neglect the potential vulnerabilities just highlighted. Subsequently, the developers of OpenSSH has fixed the issue in OpenSSH

[3] The PROTOCOL file documents OpenSSH's deviations and extensions to the SSH protocol as specified in RFCs.

```
1772  if (mac && mac->enabled && mac->etm) {
1773    if ((r = mac_compute(mac, state->p_read.seqnr,
1774        sshbuf_ptr(state->input), aadlen + need,
1775        macbuf, sizeof(macbuf))) != 0)
1776      goto out;
1777  }
1778  if ((r = sshbuf_reserve(state->incoming_packet, aadlen + need,
1779      &cp)) != 0)
1780    goto out;
1781  if ((r = cipher_crypt(&state->receive_context, state->p_read.seqnr, cp,
1782      sshbuf_ptr(state->input), need, aadlen, authlen)) != 0)
1783    goto out;
1784  if ((r = sshbuf_consume(state->input, aadlen + need + authlen)) != 0)
1785    goto out;
1786  /*
1787   * compute MAC over seqnr and packet,
1788   * increment sequence number for incoming packet
1789   */
1790  if (mac && mac->enabled) {
1791    if (!mac->etm)
1792      if ((r = mac_compute(mac, state->p_read.seqnr,
1793          sshbuf_ptr(state->incoming_packet),
1794          sshbuf_len(state->incoming_packet),
1795          macbuf, sizeof(macbuf))) != 0)
1796        goto out;
1797    if (timingsafe_bcmp(macbuf, sshbuf_ptr(state->input),
1798        mac->mac_len) != 0) {
```

Figure 6.2: The OpenSSH encryption scheme SSH-Generic-EtM prior to version 7.3 did not implement a true Encrypt-then-MAC construction. Code extracted from source code file `packet_v5_2.c` of OpenSSH version 7.2. The execution flow of SSH-Generic-EtM without errors: lines `1773-1775` computes the MAC tag over the packet; lines `1781-1782` decrypts the packet; lines `1797-1798` verifies the MAC tag.

version 7.3 [70]. The modified construction, SSH-fgEtM (SSH-Fixed-Generic-EtM), is identical to SSH-Generic-EtM except that it checks the MAC before decryption. We prove security of the modified scheme SSH-fgEtM.

SSH-fgEtM is parameterised by a symmetric encryption scheme and a message authentication code and is negotiated in OpenSSH by negotiating a special version of the message authentication code (the static string representing the MAC suffixed with `-etm`). If a special MAC is negotiated alongside SSH-AES-GCM or SSH-ChaCha20-Poly1305, then the Encrypt-then-MAC behaviour is disabled. The cryptographic processing of an SSH packet using SSH-fgEtM proceeds as follows. First, the padding length field, payload and padding length field are encrypted. Then, a MAC tag is computed over the resulting packet prepended with the length field and the sequence number. Hence, the OpenSSH implementation of SSH-fgEtM does not strictly comply with Section 6.3 of [135], which mandates that the length field

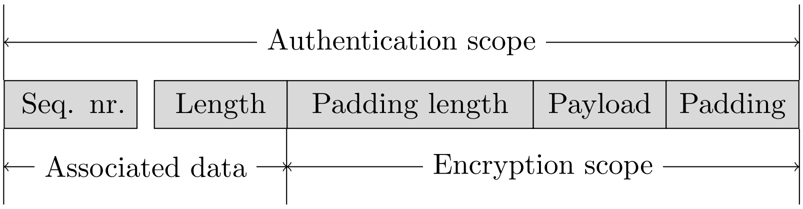

Figure 6.3: Cryptographic processing of an SSH BPP packet using the SSH encryption scheme SSH-Fixed-Generic-EtM. The size ratios between the boxes do not represent the size ratios between fields in an SSH BPP packet in general.

be encrypted. In SSH-fgEtM, this is because the MAC tag must be verified before decryption can commence, and the length field is the only indicator of where the MAC tag is located in the stream of packet bytes; so, it has to appear in unencrypted form (SSH-ChaCha20-Poly1305 circumvents this by having two different encryption contexts, see Section 6.3). Decryption proceeds similarly to SSH-ChaCha20-Poly1305, excluding the step that decrypts the length field. See Figure 6.3 for an illustration of the encryption and authentication scope when using SSH-fgEtM.

To accommodate negotiation of SSH encryption schemes that uses the MAC UMAC, we have introduced a flag use_umac. This flag is false if an SSH encryption scheme using UMAC is not negotiated and true otherwise. Note that when UMAC is negotiated, the SSH sequence number is cast to at 64-bit type. Detailed pseudo-code for SSH-fgEtM can be found in Figure 6.6 (at the end of this chapter). SSH-fgEtM is a symmetric encryption scheme supporting ciphertext fragmentation with associated key space $\mathcal{K}$ induced by the underlying choice of symmetric encryption scheme and message authentication code, plaintext space $\mathcal{M} = \mathsf{B}^*$, ciphertext space $\mathcal{C} = \mathsf{B}^*$ and error set $\mathsf{S}_\perp = \{\perp_{\mathsf{SSH\_ERR\_CONN\_CORRUPT}}, \perp_{\mathsf{SSH\_ERR\_MAC\_INVALID}}\}$.

We are now ready to state and prove our theorems showing that SSH-fgEtM is IND-sfCFA and INT-sfCTF secure. We use the notation $\mathsf{SSH\text{-}fgEtM}(\mathsf{SE}, \mathsf{MAC})$ to indicate that the algorithm pairing $(\mathsf{SE}, \mathsf{MAC})$ has been negotiated. For the proof, we make the assumption that the tagging algorithm of the message authentication code MAC is a PRF. To simplify our notation, the proof only covers the case where UMAC is not negotiated. The proofs can easily be adapted to cater for UMAC: since the nonce is derived from the SSH sequence number and hence does not repeat (assuming the number of encryptions/decryptions are restricted appropriately), its MAC tags will be pseudo-random and the proof will go through. Since the MAC tag is dependent on the sequence number, and the sequence number is incremented for each encryption/decryption, we cannot allow calling the encryption and decryption oracle more than 2^{32} times. This is captured in the following restriction:

$\mathsf{R}_{\mathsf{fgetm}}$: The adversary must make no more than 2^{32} encryption and decryption queries.

In the following, bsize is the block size of SE or equal to 8 if there is no well-defined block size of SE.

Theorem 7 (SSH-Fixed-Generic-EtM is IND-sfCFA secure)**.**
Let $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a length-preserving probabilistic symmetric encryption scheme with associated key space* $\mathcal{K}_{\mathsf{SE}} \subset \mathsf{B}^*$*, plaintext space* $\mathcal{M}_{\mathsf{SE}} = \mathsf{B}^*$*, and ciphertext space* $\mathcal{C} = \mathsf{B}^*$ *and let* $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac})$ *be a message authentication code with key space* $\mathcal{K}_{\mathsf{MAC}} \subset \mathsf{B}^*$*, message space* $\mathcal{M}_{\mathsf{MAC}} = \mathsf{B}^*$ *and tag length* ℓ_{tag}*. Let* $\mathsf{SSH\text{-}fgEtM(SE, MAC)}$ *be the SSH encryption scheme defined in Figure 6.6. Then for any* IND-sfCFA *adversary* $\mathcal{A}_{\mathsf{sfcfa}}$*, respecting restriction* $\mathsf{R}_{\mathsf{fgetm}}$*, against* $\mathsf{SSH\text{-}fgEtM(SE, MAC)}$*, there exists an* IND-CPA *adversary* $\mathcal{A}_{\mathsf{cpa}}$ *against* SE*, a* PRF *adversary* $\mathcal{A}_{\mathsf{prf}}$ *against* Mac *and an* UF-CMA *adversary* $\mathcal{A}_{\mathsf{ufcma}}$ *against* MAC *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ind\text{-}sfcfa}}_{\mathsf{SSH\text{-}fgEtM(SE,MAC)}}(\mathcal{A}_{\mathsf{sfcfa}}) \leq \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{SE}}(\mathcal{A}_{\mathsf{cpa}}) + 2 \cdot \mathsf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathsf{MAC}}(\mathcal{A}_{\mathsf{ufcma}})& \\ + 2 \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{Mac}}(\mathcal{A}_{\mathsf{prf}}),& \qquad (6.22)\end{aligned}$$

where adversaries $\mathcal{A}_{\mathsf{cpa}}$*,* $\mathcal{A}_{\mathsf{prf}}$ *and* $\mathcal{A}_{\mathsf{ufcma}}$ *have similar running times to* $\mathcal{A}_{\mathsf{sfcfa}}$*. If* $\mathcal{A}_{\mathsf{sfcfa}}$ *makes* q_e *encryption queries totalling* μ_e *bytes and* q_d *decryption queries, then:*

$\mathcal{A}_{\mathsf{ufcma}}$: *Makes* q_e *tagging oracle queries totalling at most* $\frac{\mu_e}{2} + q_e \cdot (12 + \mathsf{bsize})$ *bytes, and* 1 *verification oracle query.*

$\mathcal{A}_{\mathsf{prf}}$: *Makes* q_e *PRF oracle queries.*

$\mathcal{A}_{\mathsf{cpa}}$: *Makes* q_e *encryption oracle queries totalling at most* $\mu_e + q_e \cdot (4 + \mathsf{bsize})$ *bytes.*

If UMAC is used, the total bytes consumed by $\mathcal{A}_{\mathsf{ufcma}}$ and $\mathcal{A}_{\mathsf{prf}}$ is increased by $q_e \cdot 12$ because of the inclusion of the 8-byte nonce and the cast of the sequence number to an 8-byte type.

Proof of Theorem 7. We prove the theorem through a sequence of games. For each of these games, let WIN represent the event that the adversary guesses the bit b correctly.

G0 This is the IND-sfCFA game instantiated with SSH-fgEtM(SE, MAC). Hence:

$$\Pr\left[\mathsf{INI} : \mathcal{A}_{\mathsf{sfcfa}}^{\mathsf{LR}(b,\cdot,\cdot),\mathsf{DEC}(\cdot)} = b\right] = \Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{sfcfa}}) : \mathsf{WIN}]\,. \tag{6.23}$$

G1 In this game, we modify the decryption algorithm. Define the list $\mathcal{L}_{\mathsf{P}}$ to contain $\mathsf{frag}[0 : 4 + \ell_{\mathsf{packet}} + \ell_{\mathsf{tag}} - 1]_{\mathsf{B}}$ at index seqnr, assigned directly before the computation of the expected MAC tag τ_{expected} i.e. $\mathcal{L}_{\mathsf{P}}[\mathsf{seqnr}] = \mathsf{frag}[0 : 4 + \ell_{\mathsf{packet}} + \ell_{\mathsf{tag}} - 1]$. The entry $\mathcal{L}_{\mathsf{P}}[j-1]$ contains the jth fully assembled packet. Next, remove the computation of the expected MAC tag, and replace the verification of the expected MAC tag against the tag contained in the packet, with the condition:

$$(||(\mathcal{L}_{\mathsf{P}}) \npreceq ||(\mathcal{L}_{\mathsf{C}})) \wedge (||(\mathcal{L}_{\mathsf{P}}[0 : \mathsf{seqnr} - 1]) \preceq ||(\mathcal{L}_{\mathsf{C}}))\,. \tag{6.24}$$

That is, line 27 through line 32 in the decryption algorithm in Figure 6.6 are removed while the condition in the if-statement on line 33 is replaced by condition (6.24). Define τ_j to be the MAC tag contained in packet $\mathcal{L}_{\mathsf{P}}[j]$ and $\tau^j_{\mathsf{expected}}$ the MAC tag originally computed in the decryption algorithm. Define **BAD** to be the event that there exists j such that:

$$||(\mathcal{L}_{\mathsf{P}}[0 : j]) \npreceq ||(\mathcal{L}_{\mathsf{C}}) \wedge ||(\mathcal{L}_{\mathsf{P}}[0 : j-1]) \preceq ||(\mathcal{L}_{\mathsf{C}}) \wedge \left(\tau_j = \tau^j_{\mathsf{expected}}\right).$$

Note that such a j must necessarily be unique when combining the first two conditions. We next argue that games G0 and G1 are identical until event **BAD** and then proceed to show that event **BAD** implies a forgery in the sense of UF-CMA.

If the condition (6.24) is false, then we have either $||(\mathcal{L}_{\mathsf{P}}) \preceq ||(\mathcal{L}_{\mathsf{C}})$ or $||(\mathcal{L}_{\mathsf{P}}[0 : \mathsf{seqnr} - 1]) \npreceq ||(\mathcal{L}_{\mathsf{C}})$. If the former is true, then the games are clearly identical. The latter cannot be true, because it would imply that the condition was true in an earlier decryption algorithm call, which would set the CLOSED flag.

If the condition (6.24) is true, and $\tau_j \neq \tau^j_{\mathsf{expected}}$ then the games are again identical. In Game G0 $\tau_j \neq \tau^j_{\mathsf{expected}}$ can only happen when condition (6.24) is true because, otherwise, the computation is over in-sync packets, which would produce the correct MAC tag. Hence, in both games, the content of the output buffer msg is identical at any given point in time as long as **BAD** has not occurred. Therefore:

$$\Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{sfcfa}}) : \mathsf{WIN}] - \Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{sfcfa}}) : \mathsf{WIN}] \leq \Pr[\mathbf{BAD}]\,. \tag{6.25}$$

We now bound the probability of event **BAD**, by showing that if it occurs, it is possible to produce a valid forgery in the sense of UF-CMA. We will analyse this probability with respect to Game G0.
Given the adversary $\mathcal{A}_{\mathsf{sfcfa}}$, we build an adversary $\mathcal{A}_{\mathsf{ufcma}}$ against MAC as follows: $\mathcal{A}_{\mathsf{ufcma}}$ runs $\mathcal{A}_{\mathsf{sfcfa}}$ and simulates Game G0 but uses its own tagging oracle access to compute tags. For each decryption query, $\mathcal{A}_{\mathsf{ufcma}}$ checks condition (6.24) and, if true, submits

$$(\langle \mathsf{seqnr} \rangle_{32} \parallel \langle \ell_{\mathsf{packet}} \rangle_{32} \parallel \mathcal{L}_{\mathsf{P}}[j][4 : 4 + \ell_{\mathsf{packet}} - 1], \tau)$$

to the verification oracle, as its forgery attempt, instead of computing the tag $\tau^{j}_{\mathsf{expected}}$. τ is the extracted MAC tag and ℓ_{packet} is computed as in Game G0. j is the unique integer such that condition (6.24) is true. We next argue that this constitutes a valid forgery, establishing (6.25).
Because of condition (6.24), we must have either $\mathsf{seqnr} > |\mathcal{L}_{\mathsf{C}}|$ or $\mathcal{L}_{\mathsf{P}}[j] \neq \mathcal{L}_{\mathsf{C}}[j]$. In the former case, the number of decryptions is ahead of the number of encryptions. This means that no MAC tag can have been computed over any prior packet together with the sequence number seqnr. If this is not the case, $\mathcal{L}_{\mathsf{P}}[j]$ must be different from the jth encryption oracle output. Hence, in both cases, as long as **BAD** occurs, we have constructed a valid forgery. Thus:

$$\Pr[\mathbf{BAD}] \leq \mathsf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathsf{MAC}}(\mathcal{A}_{\mathsf{ufcma}}). \tag{6.26}$$

G2 In this game, we modify game G1 in the following way: Remove the decryption algorithm (Dec) call, subsequent padding check, and the assignment to the buffer msg from the decryption oracle. This corresponds to not perform the operations in line 37 through line 43 in Figure 6.6 when handling decryption queries.
We now argue that Game G1 and Game G2 are identical, which would imply:

$$\Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{sfcfa}}) : \mathsf{WIN}] = \Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{sfcfa}}) : \mathsf{WIN}]. \tag{6.27}$$

The only potential for difference comes from the operations after the replaced MAC tag verification. If condition (6.24) is not true, we have $||(\mathcal{L}_{\mathsf{P}}) \preceq ||(\mathcal{L}_{\mathsf{C}})$ (see above). Hence, in Game G1 the only input to Dec is in-sync packets produced legitimately by the encryption oracle. This will not produce any output to the adversary because it will be filtered out by the decryption oracle. This establishes (6.27).

G3 In this game, we modify how we compute the MAC tag. In the encryption oracle the MAC tag is not computed using the tagging algorithm Mac but, instead, by sampling a uniformly random string of length ℓ_{tag}, which is then used as the MAC tag. We argue there exists an adversary $\mathcal{A}_{\mathsf{prf}}$ such that:

$$\Pr[\mathsf{G2}(\mathcal{A}_{\mathsf{sfcfa}}) : \mathsf{WIN}] - \Pr[\mathsf{G3}(\mathcal{A}_{\mathsf{sfcfa}}) : \mathsf{WIN}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{Mac}}(\mathcal{A}_{\mathsf{prf}}). \tag{6.28}$$

Recall that a PRF adversary has access to an oracle $\mathcal{O}_{\mathsf{prf}}$ that either computes using the algorithm Mac or a randomly chosen function f. Given $\mathcal{A}_{\mathsf{sfcfa}}$, we construct $\mathcal{A}_{\mathsf{prf}}$ as follows: $\mathcal{A}_{\mathsf{prf}}$ runs $\mathcal{A}_{\mathsf{sfcfa}}$, simulating Game G2 but replacing the computation of the MAC tag with a call to $\mathcal{O}_{\mathsf{prf}}$. If $\mathcal{A}_{\mathsf{sfcfa}}$ wins (i.e. event WIN occurs) then $\mathcal{A}_{\mathsf{prf}}$ outputs 1, and 0 otherwise.
If $\mathcal{O}_{\mathsf{prf}}$ is instantiated with Mac then $\mathcal{A}_{\mathsf{prf}}$ perfectly simulates Game G2. If $\mathcal{O}_{\mathsf{prf}}$ is instantiated with a random function then $\mathcal{A}_{\mathsf{prf}}$ perfectly simulates Game G3. This is only true if no query to the random function is ever repeated; the sequence number that forms part of the query ensures this (under restriction $\mathsf{R}_{\mathsf{fgetm}}$). This proves (6.28).
To finish, we show:

$$\Pr[\mathsf{G3}(\mathcal{A}_{\mathsf{sfcfa}}) : \mathsf{WIN}] \leq \frac{\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{SE}}(\mathcal{A}_{\mathsf{cpa}}) + 1}{2}. \tag{6.29}$$

Given adversary $\mathcal{A}_{\mathsf{sfcfa}}$, we construct $\mathcal{A}_{\mathsf{cpa}}$ as follows: $\mathcal{A}_{\mathsf{cpa}}$ runs $\mathcal{A}_{\mathsf{sfcfa}}$ and simulates game G3 (without sampling a bit). Calls to the left-or-right oracle by $\mathcal{A}_{\mathsf{sfcfa}}$ are handled by first preparing two packets packet_0 and packet_1 (one for each plaintext input M_0 and M_1) as defined in ssh-fgEtM-Enc. Then, $\mathcal{A}_{\mathsf{cpa}}$ submits $(\mathsf{packet}_0, \mathsf{packet}_1)$ to its left-or-right oracle available through the IND-CPA game. This produces a packet c_{packet} which is an encryption of packet_d, where d is the bit sampled in the IND-CPA game unknown to $\mathcal{A}_{\mathsf{cpa}}$. $\mathcal{A}_{\mathsf{cpa}}$ then samples ℓ_{tag} random bytes to be used as the MAC tag. $\mathcal{A}_{\mathsf{cpa}}$ uses c_{packet} concatenated with the ℓ_{tag} random bytes to answer the encryption query from $\mathcal{A}_{\mathsf{sfcfa}}$. $\mathcal{A}_{\mathsf{cpa}}$ outputs the guess returned by $\mathcal{A}_{\mathsf{sfcfa}}$, say d'. $\mathcal{A}_{\mathsf{cpa}}$ perfectly simulates game G3 for $\mathcal{A}_{\mathsf{sfcfa}}$. If $\mathcal{A}_{\mathsf{sfcfa}}$ wins, then d' must be the correct guess of bit d ($M_{d'}$ was used to produce the packet c_{packet}). $\mathcal{A}_{\mathsf{cpa}}$ can simulate decryption queries by $\mathcal{A}_{\mathsf{sfcfa}}$ because neither Dec or MAC are needed. This proves (6.29).

Combining (6.23), (6.25), (6.26), (6.27), (6.28) and (6.29) yields the advantage bound (6.22).

Finally, we analyse the resource consumption of the adversaries. Suppose $\mathcal{A}_{\mathsf{sfcfa}}$ makes q_e encryption queries totalling μ_e bytes, and q_d decryption queries. Then:

$\mathcal{A}_{\mathsf{ufcma}}$**:** Makes q_e tagging oracle queries totalling at most $\frac{\mu_e}{2} + q_e \cdot (1 + \mathsf{bsize} + 3) + q_e \cdot (4 + 4)$ bytes, and 1 verification oracle query. The first term follows because the encryption oracle picks one out of two plaintexts. The second term follows because the plaintext is encoded with a padding length field of 1 byte and at most $\mathsf{bsize} + 3$ bytes of padding. The last term follows because for each query the additional input to the MAC tagging algorithm (Mac), excluding the packet, is a 4-byte sequence number and a 4-byte length field (assuming UMAC is not negotiated).

$\mathcal{A}_{\mathsf{prf}}$**:** Makes one oracle query for each encryption query, which makes q_e oracle queries in total.

$\mathcal{A}_{\mathsf{cpa}}$**:** Makes q_e oracle queries totalling at most $\mu_e + q_e \cdot (1 + \mathsf{bsize} + 3)$ bytes. Same arguments as above, except $\mathcal{A}_{\mathsf{cpa}}$ uses both plaintexts queried by $\mathcal{A}_{\mathsf{sfcfa}}$ which removes the denominator in the first term.

$\mathcal{A}_{\mathsf{ufcma}}$, $\mathcal{A}_{\mathsf{prf}}$ and $\mathcal{A}_{\mathsf{cpa}}$ clearly have similar running time to $\mathcal{A}_{\mathsf{sfcfa}}$. □

Theorem 8 ($\mathsf{SSH\text{-}Fixed\text{-}Generic\text{-}EtM}$ is $\mathsf{INT\text{-}sfCTF}$ secure)**.**
Let $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a length-preserving probabilistic symmetric encryption scheme with associated key space* $\mathcal{K}_{\mathsf{SE}} \subset \mathsf{B}^*$*, plaintext space* $\mathcal{M}_{\mathsf{SE}} = \mathsf{B}^*$ *and ciphertext space* $\mathcal{C} = \mathsf{B}^*$*, and let* $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac})$ *be a message authentication code with key space* $\mathcal{K}_{\mathsf{MAC}} \subset \mathsf{B}^*$*, message space* $\mathcal{M}_{\mathsf{MAC}} = \mathsf{B}^*$ *and tag length* ℓ_{tag}*. Let* $\mathsf{SSH\text{-}fgEtM(SE, MAC)}$ *be the SSH encryption scheme defined in Figure 6.6. Then for any* $\mathsf{INT\text{-}sfCTF}$ *adversary* $\mathcal{A}_{\mathsf{ctf}}$*, respecting restriction* $\mathsf{R}_{\mathsf{fgetm}}$*, against* $\mathsf{SSH\text{-}fgEtM(SE, MAC)}$*, there exists a* $\mathsf{UF\text{-}CMA}$ *adversary* $\mathcal{A}_{\mathsf{ufcma}}$ *against* MAC *such that*

$$\mathsf{Adv}^{\mathsf{ind\text{-}ctf}}_{\mathsf{SSH\text{-}fgEtM(SE,MAC)}}(\mathcal{A}_{\mathsf{ctf}}) \leq \mathsf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathsf{MAC}}(\mathcal{A}_{\mathsf{ufcma}}), \tag{6.30}$$

where $\mathcal{A}_{\mathsf{ufcma}}$ *has similar running time to* $\mathcal{A}_{\mathsf{ctf}}$*. If* $\mathcal{A}_{\mathsf{ctf}}$ *makes* q_e *encryption queries totalling* μ_e *bytes, then* $\mathcal{A}_{\mathsf{ufcma}}$ *makes* q_e *oracle queries totalling at most* $\mu_e + q_e \cdot (12 + \mathsf{bsize})$ *bytes.*

Proof of Theorem 8. The proof is almost identical to the proof for Theorem 7. For each game below, let FORGE represent the event that the adversary wins according to the win condition in Definition 21.

G0 This is the $\mathsf{INT\text{-}sfCTF}$ game instantiated with $\mathsf{SSH\text{-}fgEtM}(\mathsf{SE}, \mathsf{MAC})$. Hence:

$$\Pr\left[\mathsf{INI},\ \mathcal{A}_{\mathsf{ind\text{-}ctf}}^{\mathsf{ENC}(\cdot),\mathsf{DEC}(\cdot)}\ :\ \mathsf{FORGE}\right] = \Pr\left[\mathsf{G0}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right]. \tag{6.31}$$

G1 Make the same modifications that were made to define Game G1 in the proof of Theorem 7. Also, let **BAD** be the event defined in the same proof. Using identical arguments, we have:

$$\Pr\left[\mathsf{G0}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right] - \Pr\left[\mathsf{G1}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right] \leq \Pr\left[\mathbf{BAD}\right]. \tag{6.32}$$

Define $\mathcal{A}_{\mathsf{ufcma}}$ as in the proof for Theorem 7. Again, using identical arguments, we have:

$$\Pr\left[\mathbf{BAD}\right] \leq \mathsf{Adv}_{\mathsf{MAC}}^{\mathsf{uf\text{-}cma}}(\mathcal{A}_{\mathsf{ufcma}}). \tag{6.33}$$

G2 Make the same modifications that were made to define Game G2 in the proof of Theorem 7. Using identical arguments, we have:

$$\Pr\left[\mathsf{G1}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right] = \Pr\left[\mathsf{G2}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right]. \tag{6.34}$$

But in Game G2 it is obvious that the decryption oracle will never output any element from the set $\{0, 1, ¶\}^{+}$. Therefore:

$$\Pr\left[\mathsf{G2}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right] = 0. \tag{6.35}$$

Combining (6.31), (6.32), (6.33), (6.34), and (6.35) yields the advantage bound (6.30). $\mathcal{A}_{\mathsf{ufcma}}$ is identical to the UF-CMA adversary in the proof for Theorem 7 and consumes the same amount of resources and has similar running time. □

Note that theorems Theorem 7 and Theorem 8 only require UF-CMA. We can contrast this against a general Encrypt-then-Authenticate (the MAC is replaced with a general message authentication scheme, cf. Section 2.3.7) construction, where the message authentiation scheme used (typically) must meet SUF-CMA to be able to achieve IND-CCA - the intuition is that we must ensure the ciphertext is not malleable. But in the theorems above, we use deterministic MACs, where the two security notions UF-CMA and SUF-CMA equal strength cf. Section 2.3.7 under *security notions*).

6.4.1 SSH-Fixed-Generic-EtM and IV-based Encryption

Theorem 7 assumes SSH-fgEtM is instantiated with a probabilistic symmetric encryption scheme. This assumption does not capture instantiations with IV-based symmetric encryption schemes as defined in Definition 9. The only supported IV-based scheme in OpenSSH is CBC-mode. It is well known that such schemes are insecure if they are used in the standard Encrypt-and-MAC SSH construction. We saw several such examples in Chapter 5. The CBC construction uses *IV chaining*, where the IV is the last ciphertext block from the previous encryption, and only initially generated uniformly at random. This is generally an insecure way of implementing CBC-mode. An obvious fix would be to generate a uniformly random IV for each encryption. However, Bellare et al. [20] show that this construction is insecure (in the sense that it does not meet IND-CCA). They propose several possible modifications to the CBC-mode implementation that can yield a IND-CCA secure SSH encryption scheme. One instantiation is to use uniformly random IVs and require that the padding bytes are also chosen uniformly at random.

If we envision that SSH-fgEtM is instantiated with an IV-based symmetric encryption scheme ivSE, that generates the IV uniformly at random (and sends the IV along with the packet) and use uniformly random padding, it is possible to reuse most of the proof for Theorem 7 to show that SSH-fgEtM(ivSE, MAC) is IND-sfCFA secure with advantage bound:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ind\text{-}sfcfa}}_{\mathsf{SSH\text{-}fgEtM(ivSE,MAC)}}(\mathcal{A}_{\mathsf{sfcfa}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{ivE}}_{\mathsf{ivSE}}(\mathcal{A}_{\mathsf{ive}}) + 2 \cdot \mathsf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathsf{MAC}}(\mathcal{A}_{\mathsf{ufcma}}) \\ + 2 \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{Mac}}(\mathcal{A}_{\mathsf{prf}}). \qquad (6.36)\end{aligned}$$

The required modifications to the proof are as follows: In Game G3 don't bound the probability of winning the game. Instead, define a new Game G4 that replaces the encryption algorithm with the sampling of a random string of length $|IV|_{\mathsf{B}} + \ell_{\mathsf{packet}}$ (using that the encryption algorithm is length-preserving). Then we can bound the difference between winning G3 and G4 with the advantage of winning the ivE security game (cf. Definition 11). The proof can be finished by noting that winning Game G4 can only happen with probability 1/2.

Using the proof for Theorem 8 it can also be shown that SSH-fgEtM(ivSE, MAC) is INT-sfCTF secure with advantage bound:

$$\mathsf{Adv}^{\mathsf{ind\text{-}ctf}}_{\mathsf{SSH\text{-}fgEtM(ivSE,MAC)}}(\mathcal{A}_{\mathsf{ctf}}) \leq \mathsf{Adv}^{\mathsf{uf\text{-}cma}}_{\mathsf{MAC}}(\mathcal{A}_{\mathsf{ufcma}}).$$

Note, however, that the scheme SSH-fgEtM(ivSE, MAC) does not perfectly capture instantiating SSH-fgEtM with the OpenSSH CBC-mode encryption scheme, because the "real" SSH encryption scheme is not using random IV's.

6.5 SSH-AES-GCM in OpenSSH

The AES-GCM AEAD scheme is specified in [107]. The AES-GCM based SSH encryption scheme SSH-AES-GCM is described in RFC 5647 [85] and denoted by `aesk-gcm@openssh.com`, where k is either 128 or 256 depending on the key size. OpenSSH follows the RFC 5647 specification.

Now we turn to describe how SSH-AES-GCM is implemented in OpenSSH. Internally, OpenSSH makes use of AES-GCM encryption and decryption functions from OpenSSL. The two common inputs to these two functions are 4 bytes of additional data and a 12-byte nonce.[4] The former consists of the length field represented as a 32-bit unsigned integer. The latter is constructed by concatenating a 4-byte fixed field (fixed_field) and an 8-byte invocation counter (ICF), both of which are generated uniformly at random during setup. For each encryption and decryption operation, the ICF is incremented by one while the fixed field is invariant. Internally, the OpenSSL encryption and decryption functions also maintain a 4-byte block counter; this is initialised to one and incremented as each block of key is produced. As with SSH-Fixed-Generic-EtM (and for the same reason), SSH-AES-GCM deviates from Section 6.3 of [135] by not encrypting the length field. Additionally, SSH-AES-GCM does not follow a requirement of Section 6.4 in [135], which specifies that the sequence number must be integrity protected. Implicitly, though, this requirement is satisfied because the value of ICF is equal to the sequence number plus some fixed offset. Decryption for SSH-AES-GCM extracts the length field from the stream of received bytes, verifies the length field, and then calls the internal OpenSSL AES-GCM decryption function once sufficient bytes have arrived. See Figure 6.4 for an illustration of the encryption and authentication scope of SSH-AES-GCM in OpenSSH.

To write up a full, detailed pseudo-code of SSH-AES-GCM, we abstract the OpenSSL AES-GCM encryption and decryption functions using a nonce-based symmetric encryption scheme $\mathsf{nSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ (cf. Definition 10), with the caveat that Enc must have constant ciphertext expansion of size ℓ_{exp} bytes (necessary to locate packet boundaries). That is, if $C \leftarrow \mathsf{Enc}_{\mathsf{k}}(M)$ then $|C|_{\mathsf{B}} = |M|_{\mathsf{B}} + \ell_{\mathsf{exp}}$. This normally accounts for the addition of an authentication tag that is appended to the encrypted plaintext to produce the final ciphertext. This is, for example, true for AES-GCM

[4] The specification [85] calls this input an initialisation vector and denotes it by IV. But this is really a nonce, which is why we stick to our naming.

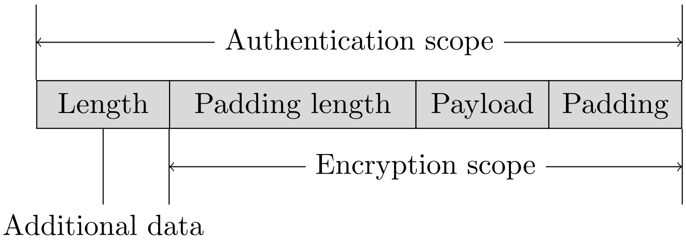

Figure 6.4: Cryptographic processing of a SSH BPP packet using the SSH encryption scheme $\mathsf{SSH\text{-}AES\text{-}GCM}$. The size ratios between the boxes do not represent the size ratios between fields in an SSH BPP packet in general.

where $\ell_{\mathsf{exp}} = 16$, which is the tag length of the MAC tag produced by the polynomial MAC GHASH used to construct AES-GCM.

Define $\mathsf{SSH\text{-}N(nSE)}$ to be the SSH encryption scheme $\mathsf{SSH\text{-}AES\text{-}GCM}$ described above, but where the AES-GCM nonce-based symmetric encryption scheme is replaced with a generic nonce-based symmetric encryption scheme nSE having 12-byte nonces. This abstraction means that Theorem 9 and Theorem 10 capture more than just $\mathsf{SSH\text{-}AES\text{-}GCM}$. $\mathsf{SSH\text{-}N}$ could be generalised further by considering a general nonce generation function that generates a new nonce for each encryption/decryption call, instead of the $\mathsf{SSH\text{-}AES\text{-}GCM}$ specific way of generating the nonce. However, for simplicity, we chose not to pursue this abstraction. See Figure 6.7 (at the end of this chapter) for the detailed pseudo-code of $\mathsf{SSH\text{-}N}$. $\mathsf{SSH\text{-}N}$ is a symmetric encryption scheme supporting ciphertext fragmentation with associated key space $\mathcal{K} = \mathsf{B}^{\mathsf{key\_len}}$, plaintext space $\mathcal{M} = \mathsf{B}^*$, ciphertext space $\mathcal{C} = \mathsf{B}^*$ and error set $\mathsf{S}_\perp = \{\perp_{\mathsf{SSH\_ERR\_CONN\_CORRUPT}}, \perp_{\mathsf{SSH\_ERR\_MAC\_INVALID}}\}$.

We are now ready to prove that $\mathsf{SSH\text{-}N}$ meets both $\mathsf{IND\text{-}sfCFA}$ and $\mathsf{INT\text{-}sfCTF}$. Because of the way the nonce is constructed and incremented (only incrementing 8 bytes of the entire 12-byte nonce), the nonce will repeat after 2^{64} increments. This means we cannot allow calling the encryption and decryption oracles more than 2^{64} times. We capture this in the following restriction:

$\mathsf{R_N}$**:** The adversary must make no more than 2^{64} encryption and decryption queries.

We first prove that $\mathsf{SSH\text{-}N}$ is $\mathsf{IND\text{-}sfCFA}$ secure under $\mathsf{R_N}$.

Theorem 9 ($\mathsf{SSH\text{-}N}$ is $\mathsf{IND\text{-}sfCFA}$ secure)**.**
Let $\mathsf{nSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a nonce-based symmetric encryption scheme with associated key space* $\mathcal{K} = \mathsf{B}^{\mathsf{key\_len}}$*, plaintext space* $\mathcal{M} = \mathsf{B}^*$*, ciphertext space* $\mathcal{C} = \mathsf{B}^*$*,*

nonce space $\mathsf{B}^{12} \subseteq \mathcal{N}$*, additional data space* $\mathsf{B}^4 \subseteq \mathcal{A}$ *and constant ciphertext expansion* ℓ_{exp}*. Let* $\mathsf{SSH\text{-}N(nSE)}$ *be the SSH encryption scheme defined in Figure 6.7. For any adversary* $\mathcal{A}_{\mathsf{sfcfa}}$*, respecting restriction* $\mathsf{R_N}$*, against* $\mathsf{SSH\text{-}N(nSE)}$*, there exists an* nAE *adversary* $\mathcal{A}_{\mathsf{nae}}$ *against* nSE *such that:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}sfcfa}}_{\mathsf{SSH\text{-}N(nSE)}}(\mathcal{A}_{\mathsf{sfcfa}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}_{\mathsf{nae}}), \tag{6.37}$$

where adversary $\mathcal{A}_{\mathsf{nae}}$ *has similar running time to* $\mathcal{A}_{\mathsf{sfcfa}}$*. If* $\mathcal{A}_{\mathsf{sfcfa}}$ *makes* q_e *encryption queries totalling* μ_e *bytes and* q_d *decryption queries, then* $\mathcal{A}_{\mathsf{nae}}$ *makes* q_e *encryption queries totalling at most* $\frac{\mu_e}{2} + q_e \cdot (8 + \mathsf{bsize})$ *and* $\frac{\mu_d}{4+\mathsf{bsize}+\ell_{\mathsf{exp}}}$ *decryption queries totalling* μ_d *bytes.*

Proof of Theorem 9. We first define two games. For each game let WIN represent the event that the adversary guesses the bit b correctly.

G0 This is the IND-sfCFA game instantiated with SSH-N(nSE). Hence:

$$\Pr\left[\mathsf{INI}\ :\ \mathcal{A}^{\mathsf{LR}(b,\cdot,\cdot),\mathsf{DEC}(\cdot)} = b\right] = \Pr\left[\mathsf{G0}(\mathcal{A}_{\mathsf{sfcfa}})\ :\ \mathsf{WIN}\right]. \tag{6.38}$$

G1 Define the list $\mathcal{L}_{\mathsf{P}}$ as in Game G1 from the proof of Theorem 7 except we assign $\mathsf{frag}[0 : 4 + \ell_{\mathsf{packet}} + \ell_{\mathsf{exp}} - 1]$ to $\mathcal{L}_{\mathsf{P}}$ at index seqnr. In the decryption algorithm ssh-n-Dec, remove line 26 through line 30 and line 35 through line 40. At the same time, replace the condition in line 31 with the condition:

$$(||(\mathcal{L}_{\mathsf{P}}) \not\preceq ||(\mathcal{L}_{\mathsf{C}})) \wedge (||(\mathcal{L}_{\mathsf{P}}[0 : \mathsf{seqnr} - 1]) \preceq ||(\mathcal{L}_{\mathsf{C}}))\,. \tag{6.39}$$

Additionally, remove the Enc algorithm in ssh-n-Enc and, instead, let c_{packet} be a uniformly random byte-string of size $\ell_{\mathsf{packet}} + \ell_{\mathsf{exp}}$ sampled per encryption query.

Given $\mathcal{A}_{\mathsf{sfcfa}}$ and nAE oracles $(\mathcal{O}_1, \mathcal{O}_2) \in \{(\mathsf{ENC}, \mathsf{DEC}), (\$, \mathsf{Err})\}$ define adversary $\mathcal{A}_{\mathsf{nae}}$ as follows: Sample a bit b and run $\mathcal{A}_{\mathsf{sfcfa}}$ answering its queries as follows:

LR: Simulate the IND-sfCFA encryption oracle LR instantiated with encryption algorithm ssh-n-Enc but replacing ssh-n-Enc with a query to $\mathcal{O}_1$ using the same input.

DEC: Simulate the IND-sfCFA decryption oracle DEC instantiated with decryption algorithm ssh-n-Dec but replacing Dec with a query to $\mathcal{O}_2$ using the same input.

If b' is the output from $\mathcal{A}_{\mathsf{sfcfa}}$, then $\mathcal{A}_{\mathsf{nae}}$ outputs 1 if $b = b'$ and 0 otherwise.

We will prove:

$$\Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] - \Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] \leq \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}_{\mathsf{nae}}). \tag{6.40}$$

If $(\mathcal{O}_1, \mathcal{O}_2) = (\mathsf{ENC}, \mathsf{DEC})$. Then $\mathcal{A}_{\mathsf{nae}}$ perfectly simulates the IND-sfCFA security game for $\mathcal{A}_{\mathsf{sfcfa}}$. Furthermore, if $\mathcal{A}_{\mathsf{sfcfa}}$ wins, then $\mathcal{A}_{\mathsf{nae}}$ returns 1, and vice versa. Hence:

$$\Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] = \Pr\left[\mathsf{INI} \,:\, \mathcal{A}_{\mathsf{nae}}^{\mathsf{ENC}(\cdot,\cdot,\cdot),\mathsf{DEC}(\cdot,\cdot,\cdot)} = 1\right]. \tag{6.41}$$

Now consider the case where $(\mathcal{O}_1, \mathcal{O}_2) = (\$, \mathsf{Err})$. In the following, we will consider the output from the decryption oracle in Game G1 and from the decryption oracle simulated by $\mathcal{A}_{\mathsf{nae}}$ when answering queries from $\mathcal{A}_{\mathsf{sfcfa}}$. We reference the former by D_{G1} and reference the latter by $D_{\mathcal{A}_{\mathsf{nae}}}$. Denote by SYNC the condition (6.39). We can consider SYNC as an event in both D_{G1} and $D_{\mathcal{A}_{\mathsf{nae}}}$. In the following, it is established that the output from D_{G1} and $D_{\mathcal{A}_{\mathsf{nae}}}$ is identical.

We first argue that if SYNC has not happened then the output in D_{G1} and $D_{\mathcal{A}_{\mathsf{nae}}}$ is identical. But since SYNC is a necessary condition for the out-of-sync flag sync to be set, this is clearly the case.

Consider now the query for which the event SYNC occurs. In D_{G1}, we have $\mathsf{msg} = \mathsf{msg}' || \perp_{\mathsf{SSH\_ERR\_MAC\_INVALID}}$ after the decryption algorithm ssh-n-Dec returns. The msg' part can only consist of in-sync plaintext produced by the encryption algorithm (otherwise SYNC would have occurred earlier). Therefore, after removing the in-sync plaintext using the list $\mathcal{L}_{\mathsf{M}}$, the output in D_{G1} is $\perp_{\mathsf{SSH\_ERR\_MAC\_INVALID}}$. The situation is the same in $D_{\mathcal{A}_{\mathsf{nae}}}$, where we have $\mathsf{msg} = \mathsf{msg}'' || \perp_{\mathsf{SSH\_ERR\_MAC\_INVALID}}$. The part msg'' can (also) only consist of in-sync plaintext. Therefore, the output in $D_{\mathcal{A}_{\mathsf{nae}}}$ is also $\perp_{\mathsf{SSH\_ERR\_MAC\_INVALID}}$.

In the event of SYNC, the closed flag CLOSED will be set in both D_{G1} and $D_{\mathcal{A}_{\mathsf{nae}}}$ and the subsequent output will therefore also be identical.

This shows that the output in D_{G1} and $D_{\mathcal{A}_{\mathsf{nae}}}$ is identical. Hence, $\mathcal{A}_{\mathsf{nae}}$ perfectly simulates Game G1. If $\mathcal{A}_{\mathsf{sfcfa}}$ win then $\mathcal{A}_{\mathsf{nae}}$ returns 1, and vice versa. Hence:

$$\Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{sfcfa}}) \,:\, \mathsf{WIN}] = \Pr\left[\mathsf{INI} \,:\, \mathcal{A}_{\mathsf{nae}}^{\$(\cdot,\cdot,\cdot),\mathsf{Err}(\cdot,\cdot,\cdot)} = 1\right]. \tag{6.42}$$

Combining (6.41) and (6.42) establishes (6.40).

Consider the outputs seen by the adversary in Game G1. The encryption oracle outputs uniformly random strings of the same length as legitimate packets. The adversary cannot learn anything from this output. The decryption oracle outputs either the empty string or the error $\perp_{\mathsf{SSH\_ERR\_MAC\_INVALID}}$. The error is only returned when the out-of-sync flag is set, which the adversary can already compute knowing the input and output of the encryption oracle. Therefore:

$$\Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{sfcfa}}) : \mathsf{WIN}] = \frac{1}{2}. \tag{6.43}$$

Combining (6.38), (6.40) and (6.43) yields advantage bound (6.37).

Suppose $\mathcal{A}_{\mathsf{sfcfa}}$ makes q_e encryption queries totalling μ_e bytes and q_d decryption queries totalling μ_d bytes. Then $\mathcal{A}_{\mathsf{nae}}$ makes q_e encryption queries totalling $\frac{\mu_e}{2} + q_e \cdot (1 + \mathsf{bsize} + 3) + q_e \cdot 4$. The first term follows because the encryption oracle picks one out of two plaintexts. The second term follows because the plaintext is encoded with a padding length field of 1 byte and at most $\mathsf{bsize} + 3$ bytes of padding. The last term follows because for each query we include 4 bytes of additional data. In addition, $\mathcal{A}_{\mathsf{nae}}$ makes at most $\frac{\mu_d}{4+\mathsf{bsize}+\ell_{\mathsf{exp}}}$ decryption queries totalling at most μ_d bytes. The number of decryption queries follows because the smallest packet size is $4 + \mathsf{bsize} + \ell_{\mathsf{exp}}$.

It can be checked that $\mathcal{A}_{\mathsf{nae}}$ has similar running time to $\mathcal{A}_{\mathsf{sfcfa}}$ □

Theorem 10 (SSH-N is INT-sfCTF secure)**.**
Let $\mathsf{nSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a nonce-based symmetric encryption scheme with associated key space* $\mathcal{K} = \mathsf{B}^{\mathsf{key\_len}}$*, plaintext space* $\mathcal{M} = \mathsf{B}^*$*, ciphertext space* $\mathcal{C} = \mathsf{B}^*$*, nonce space* $\mathsf{B}^{12} \subseteq \mathcal{N}$*, additional data space* $\mathsf{B}^4 \subseteq \mathcal{A}$ *and constant ciphertext expansion* ℓ_{exp}*. Let* $\mathsf{SSH\text{-}N(nSE)}$ *be the SSH encryption scheme defined in Figure 6.7. For any adversary* $\mathcal{A}_{\mathsf{ctf}}$*, respecting restriction* $\mathsf{R_N}$*, against* $\mathsf{SSH\text{-}N(nSE)}$*, there exists an* nAE *adversary* $\mathcal{A}_{\mathsf{nae}}$ *against* nSE *such that:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}ctf}}_{\mathsf{SSH\text{-}N(nSE)}}(\mathcal{A}_{\mathsf{ctf}}) \leq \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}_{\mathsf{nae}}), \tag{6.44}$$

where adversary $\mathcal{A}_{\mathsf{nae}}$ *has similar running time to* $\mathcal{A}_{\mathsf{ctf}}$*. If* $\mathcal{A}_{\mathsf{ctf}}$ *makes* q_e *encryption queries totalling* μ_e *bytes and* q_d *decryption queries totalling at most* μ_d *bytes, then* $\mathcal{A}_{\mathsf{nae}}$ *makes* q_e *encryption queries totalling at most* $\mu_e + q_e \cdot (8 + \mathsf{bsize})$ *and* $\frac{\mu_d}{4+\mathsf{bsize}+\ell_{\mathsf{exp}}}$ *decryption queries totalling at most* μ_d *bytes.*

Proof of Theorem 10. The proof is almost identical to the proof of Theorem 10. For each game below, let FORGE represent the event that the adversary wins according to the win condition in Definition 21.

G0 This is the $\mathsf{INT\text{-}sfCTF}$ game instantiated with $\mathsf{SSH\text{-}N(nSE)}$. Hence:

$$\Pr\left[\mathsf{INI},\ \mathcal{A}_{\mathsf{ctf}}^{\mathsf{ENC}(\cdot),\mathsf{DEC}(\cdot)}\ :\ \mathsf{FORGE}\right] = \Pr\left[\mathsf{G0}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right]. \tag{6.45}$$

G1 Make the same modifications that were made to define Game $\mathsf{G1}$ in the proof of Theorem 9. Additionally, define $\mathcal{A}_{\mathsf{nae}}$ similarly, except $\mathcal{A}_{\mathsf{nae}}$ outputs 1 if FORGE occurs and 0 otherwise. Using the exact same arguments, we have:

$$\Pr\left[\mathsf{G0}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right] \le \mathsf{Adv}_{\mathsf{nSE}}^{\mathsf{nAE}}(\mathcal{A}_{\mathsf{nae}}) + \Pr\left[\mathsf{G1}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right]. \tag{6.46}$$

But in Game $\mathsf{G1}$ it is obvious that the decryption oracle will never output any element from the set $\{0, 1, \P\}^+$. Therefore:

$$\Pr\left[\mathsf{G1}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}\right] = 0. \tag{6.47}$$

Combining (6.45), (6.46) and (6.47) yields advantage bound (6.44).

Suppose $\mathcal{A}_{\mathsf{ctf}}$ makes q_e encryption queries totalling μ_e bytes and q_d decryption queries totalling μ_d bytes. Then $\mathcal{A}_{\mathsf{nae}}$ makes q_e encryption queries totalling $\mu_e + q_e \cdot (1 + \mathsf{bsize} + 3) + q_e \cdot 4$. The first term follows because the encryption oracle uses the entire input in its own query. The second term follows because the plaintext is encoded with a padding length field of 1 byte and at most $\mathsf{bsize} + 3$ bytes of padding. The last term follows because for each query we include 4 bytes of additional data. In addition, $\mathcal{A}_{\mathsf{nae}}$ makes at most $\frac{\mu_d}{4+\mathsf{bsize}+\ell_{\mathsf{exp}}}$ decryption queries totalling at most μ_d bytes. The number of decryption queries follows because the smallest packet size is $4 + \mathsf{bsize} + \ell_{\mathsf{exp}}$.

It can be checked that $\mathcal{A}_{\mathsf{nae}}$ has similar running time to $\mathcal{A}_{\mathsf{sfcfa}}$ □

6.6 Boundary Hiding and DoS

As mentioned previously, in addition to confidentiality and integrity, the relevant SSH RFC [135] aims to mitigate against traffic analysis and denial of service. Encrypting the length field in basic SSH encryption schemes (such as schemes using CBC-mode or CTR-mode) is designed to make traffic analysis based on packet lengths more difficult for passive attackers. However, an active attacker can manipulate packet bits after the length field and observe the number of bytes injected before a MAC error is produced.[5] Denial of service here refers to an attacker flipping bits in the (encrypted)

[5] When CBC-mode is used, OpenSSH's countermeasure to [?] prevents this simple byte-counting attack. However, a timing channel still exists, cf. Chapter 5

packet length field, causing the receiver to expect a very long packet, leading to a long delay in interaction for the sender and allocation of resources on the receiver's end, cf. [114]. Indeed, the RFC states that implementations "SHOULD check that the packet length is reasonable" and OpenSSH imposes an upper limit of 2^{18} on the 32-bit packet length field. Boldyreva et al. [36] introduced formal security notions to capture these goals, namely BH-CPA, BH-sfCFA, and n-DOS-sfCFA, presented in Chapter 3. We discuss these informally in relation to SSH.

Completely hiding plaintext lengths is impossible unless some efficiency is sacrificed. Indeed, simply encrypting the length field does not conceal plaintext lengths, at least not in a sense that is easy to formalise. However, one can hope to hide (encrypted) packet boundaries, meaning that a sequence of packets will look like a stream of random bits, which can help to mitigate traffic analysis. Intuitively, the boundary hiding notions (cf. Section 3.6) say that, given a concatenation of packets, an adversary is unable to determine the packet boundaries, and hence can neither determine the number of packets included in the concatenation nor their individual sizes. Since both SSH-Fixed-Generic-EtM and SSH-AES-GCM expose the length field in the clear it is trivial for an adversary to determine packet boundaries. In contrast, in Theorem 6 we showed SSH-ChaCha20-Poly1305 meets BH-CPA. However, SSH-ChaCha20-Poly1305 is not BH-sfCFA-secure due to the bit-flipping attack outlined above.

Recall, n-DOS-sfCFA security (cf. Section 3.7) requires that no adversary be able to forge a sequence of packet fragments, totalling n bits, such that the decryption of the sequence returns no output. One can always trivially achieve n-DOS-sfCFA security by imposing an upper limit on the packet size, but this is not ideal as it would necessarily limit the maximum plaintext size. Thus, the technically interesting (and useful) case is when n is significantly smaller than the longest possible packet. None of the three SSH encryption schemes mentioned in this chapter achieve n-DOS-sfCFA for n smaller than the maximum packet size, since even though the packet length field is integrity protected in all three cases, the MAC tag is only verified after the complete packet (as indicated by the packet length field) has been received. Thus, an adversary could change the contents of the length field to the maximum accepted value (2^{18} for OpenSSH), and the receiver would experience a connection hang until it had received 256 kilobytes of packet data, at which point the connection would be dropped.

Bringing together the discussion in this section with the results of the previous

	IND-sfCFA	INT-sfCTF	BH-CPA	BH-sfCFA	n-DOS-sfCFA
SSH-ChaCha20-Poly1305 Figure 6.5	✓ Theorem 5	✓ Theorem 4	✓ Theorem 6	✗	✗
SSH-Fixed-Generic-EtM Figure 6.6	✓ Theorem 7	✓ Theorem 8	✗	✗	✗
SSH-AES-GCM Figure 6.7	✓ Theorem 9	✓ Theorem 10	✗	✗	✗

Table 6.1: Security comparison of SSH encryption schemes in OpenSSH that are analysed in Chapter 6.

sections justifies the content of Table 6.1.

alg. ssh-ChaCha20-Poly1305-Gen

```
1 :  seqnr = 0
2 :  frag = $\epsilon$
3 :  CLOSED = false
4 :  k $\leftarrow\$$ $B^{64}$
5 :  $\sigma$ = seqnr
6 :  $\varrho$ = (frag, seqnr, $\ell_{packet}$, CLOSED)
7 :  return (k, $\sigma$, $\varrho$)
```

alg. ssh-ChaCha20-Poly1305-Enc$_k(m, \sigma)$

```
1 :  seqnr ← Parse($\sigma$)
2 :  $k_1 = k[0:31]_B$
3 :  $k_2 = k[32:63]_B$
4 :  $\ell_m = |m|_B$
5 :  $\ell_{pad} = 8 - ((1 + \ell_m) \bmod 8)$
6 :  if $\ell_{pad} < 4$
7 :     $\ell_{pad} = \ell_{pad} + 8$
8 :  pad $\leftarrow\$$ $B^{\ell_{pad}}$
9 :  packet = $\langle \ell_{pad} \rangle_8 \,\|\, m \,\|$ pad
10 : $\ell_{packet} = 1 + \ell_m + \ell_{pad}$
11 : block_ctr = 0
12 : nonce = $\langle$seqnr$\rangle_{64}$
13 : $k_{poly}$ ← ChaCha20$_{k_2}$(($0x00)^{32}$, nonce, block_ctr)
14 : $c_{length}$ ← ChaCha20$_{k_1}$($\langle \ell_{packet} \rangle_{32}$, nonce, block_ctr)
15 : block_ctr = 1
16 : $c_{packet}$ ← ChaCha20$_{k_2}$(packet, nonce, block_ctr)
17 : $\tau$ ← Poly1305MAC$_{k_{poly}}$($c_{length} \,\|\, c_{packet}$)
18 : seqnr = seqnr + 1
19 : return ($c_{length} \,\|\, c_{packet} \,\|\, \tau$, seqnr)
```

ssh-ChaCha20-Poly1305-Dec$_k(f, \varrho)$

```
1 :  if CLOSED
2 :     return ($\varepsilon$, $\varrho$)
3 :  (frag, seqnr, $\ell_{packet}$, CLOSED) ← Parse($\varrho$)
4 :  $k_1 = k[0:31]_B$
5 :  $k_2 = k[32:63]_B$
6 :  frag = frag $\|$ $f$
7 :  $m = \epsilon$
8 :  msg = $\epsilon$
9 :  while $|$frag$|_B > 0$
10 :    if $\ell_{packet} = 0$
11 :       if $\ell_{frag} \geq 4$
12 :          block_ctr = 0
13 :          nonce = $\langle$seqnr$\rangle_{64}$
14 :          $c_{length}$ = frag$[0:3]_B$
15 :          $\ell_{packet} = \langle$ChaCha20$_{k_1}$($c_{length}$, nonce, block_ctr)$\rangle^{-1}$
16 :          if $\neg(5 \leq \ell_{packet} \leq 2^{18})$
17 :             CLOSED = true
18 :             $\varrho$ = (frag, seqnr, $\ell_{packet}$, CLOSED)
19 :             return (msg $\|\perp_{SSH\_ERR\_CONN\_CORRUPT}$, $\varrho$)
20 :       else
21 :          $\varrho$ = (frag, seqnr, $\ell_{packet}$, CLOSED)
22 :          return (msg, $\varrho$)
23 :       if $\ell_{packet} \bmod 8 \neq 0$
24 :          CLOSED = true
25 :          $\varrho$ = (frag, seqnr, $\ell_{packet}$, CLOSED)
26 :          return (msg $\|\perp_{SSH\_ERR\_MAC\_INVALID}$, $\varrho$)
27 :    $\ell_{frag} = |$frag$|_B$
28 :    if $\ell_{frag} < 4 + \ell_{packet} + 16$
29 :       $\varrho$ = (frag, seqnr, $\ell_{packet}$, CLOSED)
30 :       return (msg, $\varrho$)
31 :    else
32 :       $\tau$ = frag$[4 + \ell_{packet} : 4 + \ell_{packet} + 16 - 1]_B$
33 :       block_ctr = 0
34 :       nonce = $\langle$seqnr$\rangle_{32}$
35 :       $k_{poly}$ ← ChaCha20$_{k_2}$(($0x00)^{32}$, nonce, block_ctr)
36 :       $\tau_{expected}$ ← Poly1305MAC$_{k_{poly}}$(frag$[0 : 4 + \ell_{packet}]$)
37 :       if $\tau \neq \tau_{expected}$
38 :          CLOSED = true
39 :          $\varrho$ = (frag, seqnr, $\ell_{packet}$, CLOSED)
40 :          return (msg $\|\perp_{SSH\_ERR\_MAC\_INVALID}$, $\varrho$)
41 :       block_ctr = 1
42 :       $m$ ← ChaCha20$_{k_2}$(frag$[4 : 4 + \ell_{packet}]$, nonce, block_ctr)
43 :       $\ell_{pad} = \langle m[0]_B \rangle^{-1}$
44 :       if $\ell_{pad} < 4$
45 :          CLOSED = true
46 :          $\varrho$ = (frag, seqnr, $\ell_{packet}$, CLOSED)
47 :          return (msg $\|\perp_{SSH\_ERR\_CONN\_CORRUPT}$, $\varrho$)
48 :       msg = msg $\|$ $m[1 : 1 + \ell_{packet} - (1 + \ell_{pad}) - 1]_B$ $\|$ ¶
49 :       frag = frag$[4 + \ell_{packet} + 16 - 1 : \ell_{frag} - 1]_B$
50 :       seqnr = seqnr + 1
51 :       $\ell_{packet} = 0$
52 : return (msg, (frag, seqnr, $\ell_{packet}$, CLOSED))
```

Figure 6.5: Description of SSH-ChaCha20-Poly1305 in OpenSSH. The number 8 appearing in lines (5) and (7) in ssh-ChaCha20-Poly1305-Enc, and line (23) in ssh-ChaCha20-Poly1305-Dec denotes the default SSH block size (counted in bytes). The number 16 appearing in the ssh-ChaCha20-Poly1305-Dec is the length (in bytes) of the tag τ.

```
alg. ssh-fgEtM-Gen
$\mathsf{seqnr} = 0$
$\ell_{\mathsf{packet}} = 0$
$\mathsf{frag} = \epsilon$
$\mathsf{CLOSED} = \mathsf{false}$
$\mathsf{k}_e \leftarrow \mathsf{Gen}_\mathsf{e}$
$\mathsf{k}_m \leftarrow \mathsf{Gen}_\mathsf{m}$
$\mathsf{k} = \mathsf{k}_e \parallel \mathsf{k}_m$
$\sigma = \mathsf{seqnr}$
$\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{CLOSED})$
return $(\mathsf{k}, \sigma, \varrho)$
```

```
alg. ssh-fgEtM-Enc_k(m, σ)
$\mathsf{seqnr} \leftarrow \mathsf{Parse}(\sigma)$
$\mathsf{k}_e \parallel \mathsf{k}_m \leftarrow \mathsf{Parse}(\mathsf{k})$
$\ell_\mathsf{m} = |m|_\mathsf{B}$
$\ell_\mathsf{pad} = \mathsf{bsize} - ((1 + \ell_\mathsf{m}) \bmod \mathsf{bsize})$
if $\ell_\mathsf{pad} < 4$
   $\ell_\mathsf{pad} = \ell_\mathsf{pad} + \mathsf{bsize}$
$\mathsf{pad} \leftarrow\!\$\ \mathsf{B}^{\ell_\mathsf{pad}}$
$\mathsf{packet} = \langle \ell_\mathsf{pad} \rangle_8 \parallel m \parallel \mathsf{pad}$
$\ell_\mathsf{packet} = 1 + \ell_\mathsf{m} + \ell_\mathsf{pad}$
$c_\mathsf{packet} \leftarrow \mathsf{Enc}_{\mathsf{k}_e}(\mathsf{packet})$
if $\mathsf{use\_umac}$
   $\tau \leftarrow \mathsf{Mac}_{\mathsf{k}_m}(\langle \mathsf{seqnr} \rangle_{64} \parallel \langle \ell_\mathsf{packet} \rangle_{32} \parallel c_\mathsf{packet},$
                  $\langle \mathsf{seqnr} \rangle_{64})$
else
   $\tau \leftarrow \mathsf{Mac}_{\mathsf{k}_m}(\langle \mathsf{seqnr} \rangle_{32} \parallel \langle \ell_\mathsf{packet} \rangle_{32} \parallel c_\mathsf{packet})$
$\mathsf{seqnr} = \mathsf{seqnr} + 1$
return $(\langle \ell_\mathsf{packet} \rangle_{32} \parallel c_\mathsf{packet} \parallel \tau, \mathsf{seqnr})$
```

```
alg. ssh-fgEtM-Dec_k(f, ϱ)
if $\mathsf{CLOSED}$
   return $(\varepsilon, \varrho)$
$(\mathsf{frag}, \mathsf{seqnr}, \ell_\mathsf{packet}, \mathsf{CLOSED}) \leftarrow \mathsf{Parse}(\varrho)$
$\mathsf{k}_e \parallel \mathsf{k}_m \leftarrow \mathsf{Parse}(\mathsf{k})$
$\mathsf{frag} = \mathsf{frag} \parallel f$
$m = \epsilon$
while $|\mathsf{frag}|_\mathsf{B} > 0$
   if $\ell_\mathsf{packet} = 0$
      if $\mathsf{frag} \geq 4$
         $\ell_\mathsf{packet} = \langle \mathsf{frag}[0:3]_\mathsf{B} \rangle^{-1}$
         if $\neg\,(5 \leq \ell_\mathsf{packet} \leq 2^{18})$
            $\mathsf{CLOSED} = \mathsf{true}$
            $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_\mathsf{packet}, \mathsf{CLOSED})$
            return $(\mathsf{msg} \parallel \bot_{\mathsf{SSH\_ERR\_CONN\_CORRUPT}}, \varrho)$
      else
         $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_\mathsf{packet}, \mathsf{CLOSED})$
         return $(\mathsf{msg}, \varrho)$
      if $\ell_\mathsf{packet} \bmod \mathsf{bsize} \neq 0$
         $\mathsf{CLOSED} = \mathsf{true}$
         $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_\mathsf{packet}, \mathsf{CLOSED})$
         return $(\mathsf{msg} \parallel \bot_{\mathsf{SSH\_ERR\_MAC\_INVALID}}, \varrho)$
   $\ell_\mathsf{frag} = |\mathsf{frag}|_\mathsf{B}$
   if $\ell_\mathsf{frag} < 4 + \ell_\mathsf{packet} + \ell_\mathsf{tag}$
      $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_\mathsf{packet}, \mathsf{CLOSED})$
      return $(\mathsf{msg}, \varrho)$
   else
      if $\mathsf{use\_umac}$
         $\tau_\mathsf{expected} \leftarrow \mathsf{Mac}_{\mathsf{k}_m}(\langle \mathsf{seqnr} \rangle_{64} \parallel \langle \ell_\mathsf{packet} \rangle_{32} \parallel \mathsf{frag}[4 : 4 + \ell_\mathsf{packet} - 1]_\mathsf{B},$
                                 $\langle \mathsf{seqnr} \rangle_{64})$
      else
         $\tau_\mathsf{expected} \leftarrow \mathsf{Mac}_{\mathsf{k}_m}(\langle \mathsf{seqnr} \rangle_{32} \parallel \langle \ell_\mathsf{packet} \rangle_{32} \parallel \mathsf{frag}[4 : 4 + \ell_\mathsf{packet} - 1]_\mathsf{B})$
      $\tau = \mathsf{frag}[4 + \ell_\mathsf{packet} : 4 + \ell_\mathsf{packet} + \ell_\mathsf{tag} - 1]_\mathsf{B}$
      if $\tau \neq \tau_\mathsf{expected}$
         $\mathsf{CLOSED} = \mathsf{true}$
         $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_\mathsf{packet}, \mathsf{CLOSED})$
         return $(\mathsf{msg} \parallel \bot_{\mathsf{SSH\_ERR\_MAC\_INVALID}}, \varrho)$
      $m \leftarrow \mathsf{Dec}_{\mathsf{k}_e}(\mathsf{frag}[4 : 4 + \ell_\mathsf{packet} - 1]_\mathsf{B})$
      $\ell_\mathsf{pad} = \langle m[0]_\mathsf{B} \rangle^{-1}$
      if $\ell_\mathsf{pad} < 4$
         $\mathsf{CLOSED} = \mathsf{true}$
         $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_\mathsf{packet}, \mathsf{CLOSED})$
         return $(\mathsf{msg} \parallel \bot_{\mathsf{SSH\_ERR\_CONN\_CORRUPT}}, \varrho)$
      $\mathsf{msg} = \mathsf{msg} \parallel m[1 : 1 + \ell_\mathsf{packet} - (1 + \ell_\mathsf{pad}) - 1]_\mathsf{B} \parallel \P$
      $\mathsf{frag} = \mathsf{frag}[4 + \ell_\mathsf{packet} + \ell_\mathsf{tag} - 1 : \ell_\mathsf{frag} - 1]_\mathsf{B}$
      $\mathsf{seqnr} = \mathsf{seqnr} + 1$
      $\ell_\mathsf{packet} = 0$
return $(\mathsf{msg}, (\mathsf{frag}, \mathsf{seqnr}, \ell_\mathsf{packet}, \mathsf{CLOSED}))$
```

Figure 6.6: Description of SSH-Fixed-Generic-EtM in OpenSSH instantiated with the symmetric encryption scheme SE and message authentication code MAC. The parameter bsize denotes the block size of the symmetric encryption scheme SE. If such a block size is not well-defined, SSH defaults to a block size of 8 (counted in bytes), as explained in Section 6.1.

ssh-n-Gen

```
$\mathsf{seqnr} = 0$
$\ell_{\mathsf{packet}} = 0$
$\mathsf{frag} = \epsilon$
$\mathsf{CLOSED} = \mathsf{false}$
$\mathsf{k} \leftarrow\!\!\$\ \mathsf{B}^{\mathsf{key\_len}}$
$\mathsf{ICF} \leftarrow\!\!\$\ \mathsf{B}^4$
$\mathsf{fixed\_field} \leftarrow\!\!\$\ \mathsf{B}^8$
$\mathsf{nonce} = \mathsf{fixed\_field} \parallel \mathsf{ICF}$
$\sigma = \mathsf{nonce}$
$\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{nonce}, \mathsf{CLOSED})$
return $(\mathsf{k}, \sigma, \varrho)$
```

ssh-n-Enc$_{\mathsf{k}}(m, \sigma)$

```
$\mathsf{nonce} \leftarrow \mathsf{Parse}(\sigma)$
$\ell_{\mathsf{m}} = |m|_{\mathsf{B}}$
$\ell_{\mathsf{pad}} = \mathsf{bsize} - ((1 + \ell_{\mathsf{m}}) \bmod \mathsf{bsize})$
if $\ell_{\mathsf{pad}} < 4$
  $\ell_{\mathsf{pad}} = \ell_{\mathsf{pad}} + \mathsf{bsize}$
$\mathsf{pad} \leftarrow\!\!\$\ \mathsf{B}^{\ell_{\mathsf{pad}}}$
$\mathsf{packet} = \langle \ell_{\mathsf{pad}} \rangle_8 \parallel m \parallel \mathsf{pad}$
$\ell_{\mathsf{packet}} = 1 + \ell_{\mathsf{m}} + \ell_{\mathsf{pad}}$
$\mathsf{add} = \langle \ell_{\mathsf{packet}} \rangle_{32}$
$\mathsf{fixed\_field} = \mathsf{nonce}[0:3]_{\mathsf{B}}$
$\mathsf{ICF} = \mathsf{nonce}[4:11]_{\mathsf{B}}$
$\mathsf{nonce} = \mathsf{fixed\_field} \parallel \langle \langle \mathsf{ICF} \rangle^{-1} + 1 \rangle_{64}$
$c_{\mathsf{packet}} \leftarrow \mathsf{Enc}_{\mathsf{k}}(\mathsf{packet}, \mathsf{nonce}, \mathsf{add})$
return $(\langle \ell_{\mathsf{packet}} \rangle_{32} \parallel c_{\mathsf{packet}}, \mathsf{nonce})$
```

ssh-n-Dec$_{\mathsf{k}}(f, \varrho)$

```
if CLOSED
  return $(\varepsilon, \varrho)$
$(\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{nonce}, \mathsf{CLOSED}) \leftarrow \mathsf{Parse}(\varrho)$
$\mathsf{frag} = \mathsf{frag} \parallel f$
$m = \epsilon$
while $|\mathsf{frag}|_{\mathsf{B}} > 0$
  if $\ell_{\mathsf{packet}} = 0$
    if $\mathsf{frag} \geq 4$
      $\ell_{\mathsf{packet}} = \langle \mathsf{frag}[0:3]_{\mathsf{B}} \rangle^{-1}$
      if $\neg\left(5 \leq \ell_{\mathsf{packet}} \leq 2^{18}\right)$
        $\mathsf{CLOSED} = \mathsf{true}$
        $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{nonce}, \mathsf{CLOSED})$
        return $(\mathsf{msg} \parallel \bot_{\mathsf{SSH\_ERR\_CONN\_CORRUPT}}, \varrho)$
    else
      $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{nonce}, \mathsf{CLOSED})$
      return $(\mathsf{msg}, \varrho)$
    if $\ell_{\mathsf{packet}} \bmod \mathsf{bsize} \neq 0$
      $\mathsf{CLOSED} = \mathsf{true}$
      $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{nonce}, \mathsf{CLOSED})$
      return $(\mathsf{msg} \parallel \bot_{\mathsf{SSH\_ERR\_MAC\_INVALID}}, \varrho)$
  $\ell_{\mathsf{frag}} = |\mathsf{frag}|_{\mathsf{B}}$
  if $\ell_{\mathsf{frag}} < 4 + \ell_{\mathsf{packet}} + \ell_{\mathsf{exp}}$
    $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{nonce}, \mathsf{CLOSED})$
    return $(\mathsf{msg}, \varrho)$
  else
    $\mathsf{fixed\_field} = \mathsf{nonce}[0:3]_{\mathsf{B}}$
    $\mathsf{ICF} = \mathsf{nonce}[4:11]_{\mathsf{B}}$
    $\mathsf{nonce} = \mathsf{fixed\_field} \parallel \langle \langle \mathsf{ICF} \rangle^{-1} + 1 \rangle_{64}$
    $m \leftarrow\!\!\$\ \mathsf{Dec}_{\mathsf{k}}(\mathsf{frag}[4 : 4 + \ell_{\mathsf{packet}} + \ell_{\mathsf{exp}} - 1]_{\mathsf{B}}, \mathsf{nonce},$
                  $\mathsf{frag}[0:3]_{\mathsf{B}})$
    if $m = \bot$
      $\mathsf{CLOSED} = \mathsf{true}$
      $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{nonce}, \mathsf{CLOSED})$
      return $(\mathsf{msg} \parallel \bot_{\mathsf{SSH\_ERR\_MAC\_INVALID}}, \varrho)$
    $\ell_{\mathsf{pad}} = \langle m[0]_{\mathsf{B}} \rangle^{-1}$
    if $\ell_{\mathsf{pad}} < 4$
      $\mathsf{CLOSED} = \mathsf{true}$
      $\varrho = (\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{nonce}, \mathsf{CLOSED})$
      return $(\mathsf{msg} \parallel \bot_{\mathsf{SSH\_ERR\_CONN\_CORRUPT}}, \varrho)$
    $\mathsf{msg} = \mathsf{msg} \parallel m[1 : 1 + \ell_{\mathsf{packet}} - (1 + \ell_{\mathsf{pad}}) - 1]_{\mathsf{B}} \parallel \P$
    $\mathsf{frag} = \mathsf{frag}[4 + \ell_{\mathsf{packet}} + \ell_{\mathsf{exp}} - 1 : \left|\ell_{\mathsf{frag}}\right|_{\mathsf{B}} - 1]_{\mathsf{B}}$
    $\mathsf{seqnr} = \mathsf{seqnr} + 1$
    $\ell_{\mathsf{packet}} = 0$
return $(\mathsf{msg}, (\mathsf{frag}, \mathsf{seqnr}, \ell_{\mathsf{packet}}, \mathsf{nonce}, \mathsf{CLOSED}))$
```

Figure 6.7: Description of SSH-N. The parameter bsize denotes the block size of the symmetric encryption scheme SE. If such a block size is not well-defined, SSH defaults to a block size of 8, as explained in Section 6.1.

Chapter 7

An Improved SSH Encryption Scheme

In this chapter, we look to fix the weaknesses identified in Chapter 6 by implementing a symmetric encryption scheme supporting ciphertext fragmentation that meets all four security notions in the ciphertext fragmentation model. The scheme in focus is InterMAC, first defined by Boldyreva et al. [36]. We modify the scheme to make it practical, analyse the modified scheme in the ciphertext fragmentation model, describe a library libInterMAC *that implements InterMAC, and, finally, we construct InterMAC-based SSH encryption schemes in OpenSSH using the library.*

This chapter also highlights many practical aspects of implementing a cryptographic algorithm from scratch, from considering side-channel protections to the practical performance of the algorithm in different use-cases, and what can be done to improve performance.

7.1 Weaknesses in SSH Encryption Schemes

SSH provides confidentiality and integrity through symmetric encryption and message authentication codes and in Chapter 6 we proved that several SSH encryption schemes in OpenSSH provide these security properties in the ciphertext fragmentation model. However, we also saw that no scheme has protection against active boundary hiding or denial-of-service attacks. The newly introduced schemes SSH-AES-GCM and SSH-Generic-EtM even fall short of a passive boundary hiding adversary, because the length field is readily available in the clear in the packet. This goes against one of the main security goals of SSH to hide plaintext sizes. The newest SSH encryption scheme in OpenSSH is SSH-ChaCha20-Poly1305 [111]. This scheme reintroduces the encryption of the length field via a construction that uses separate encryption keys for the length field and the rest of the data. The effort expended to encrypt the length field here highlights the importance given to hiding message plaintext by the OpenSSH developers.

7.2 InterMAC

A symmetric encryption scheme that simultaneously meets $\mathsf{IND\text{-}sfCFA}$, $\mathsf{INT\text{-}sfCTF}$, $\mathsf{BH\text{-}sfCFA}$ and $\mathsf{DOS\text{-}sfCFA}$ is the InterMAC scheme from [36]. The construction is straightforward and, in simple terms, can be described as follows. A message is split into equal-sized chunks which are then individually fed into an Encrypt-then-MAC construction. The resulting ciphertexts and MAC tags are concatenated to form the final ciphertext. We will make various modifications to InterMAC, with the main changes being:

- Extending the original InterMAC scheme to support arbitrary length, byte-oriented messages.
- Replacing the Encrypt-then-MAC construction with a general, nonce-based AE scheme.

We prove that the modifications made to InterMAC do not change its security properties, see Section 7.2.3.

7.2.1 Original InterMAC

Definition 25 below defines the original InterMAC scheme appearing in [36]. Note that the chunk length N, message sizes, chunk lengths, etc., are all counted in bytes compared to the original presentation which counted in bits. This is consistent with us adopting a presentation of the scheme as explained in Section 2.2.

Definition 25.
Let $\mathsf{SE} = (\mathsf{Gen_e}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric encryption scheme with an associated plaintext space that contains* B^{N+1}*, for some desired* $N \in \mathbb{N}$*. Furthermore, assume that* Enc *encrypts all messages of length* $N + 1$ *bytes to ciphertexts of* ℓ_{c} *bytes. Let* $\mathsf{MAC} = (\mathsf{Gen_m}, \mathsf{Tag})$ *be a message authentication code with associated plaintext space* $\{0, 1\}^*$ *and tag length* ℓ_{tag} *(counted in bytes). Then the (byte-oriented) InterMAC*

alg. $\overline{\mathsf{Gen}}$

```
$\mathsf{k}_e \leftarrow \mathsf{Gen_e}$
$\mathsf{k}_m \leftarrow \mathsf{Gen_m}$
$\mathsf{k} = \mathsf{k}_e \parallel \mathsf{k}_m$
$\mathsf{fail} = \mathsf{false}$
$\mathsf{msg\_ctr} = 0$
$\mathsf{chunk\_ctr} = 0$
$\sigma = \mathsf{msg\_ctr}$
$\varrho = (\epsilon, \epsilon, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail})$
return $(\mathsf{k}, \sigma, \varrho)$
```

alg. $\overline{\mathsf{Enc}}_\mathsf{k}(m, \sigma)$

```
$\mathsf{k}_e \parallel \mathsf{k}_m \leftarrow \mathsf{Parse}(\mathsf{k})$
$\mathsf{msg\_ctr} \leftarrow \mathsf{Parse}(\sigma)$
$c = \epsilon$
$b = \mathtt{0x00}$
for $\mathsf{chunk\_ctr} = 0 \ldots \left(\frac{|m|_\mathsf{B}}{N} - 1\right)$
    $l = \mathsf{chunk\_ctr} \cdot N$
    $r = (\mathsf{chunk\_ctr} + 1) \cdot N - 1$
    $m_\mathsf{chunk} = m[l : r]_\mathsf{B}$
    if $q = |m|_\mathsf{B}$
        $b = \mathtt{0x01}$
    $c_\mathsf{chunk} \leftarrow \mathsf{Enc}_{\mathsf{k}_e}(m_\mathsf{chunk} \parallel b)$
    $\tau \leftarrow \mathsf{Mac}_{\mathsf{k}_m}(c_\mathsf{chunk} \parallel \langle \mathsf{msg\_ctr} \rangle \parallel$
                    $\langle \mathsf{chunk\_ctr} \rangle)$
    $c = c \parallel c_\mathsf{chunk} \parallel \tau$
$\mathsf{msg\_ctr} = \mathsf{msg\_ctr} + 1$
return $(c, \mathsf{msg\_ctr})$
```

alg. $\overline{\mathsf{Dec}}_\mathsf{k}(f, \varrho)$

```
$\mathsf{k}_e \parallel \mathsf{k}_m \leftarrow \mathsf{Parse}(\mathsf{k})$
$(\mathsf{frag}, \mathsf{msg}, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail}) \leftarrow \mathsf{Parse}(\varrho)$
$\mathsf{frag} = \mathsf{frag} \parallel f$
$m = \epsilon$
while $|\mathsf{frag}|_\mathsf{B} \geq \ell_\mathsf{c} + \ell_\mathsf{tag}$
    $c_\mathsf{chunk} = \mathsf{frag}[0 : \ell_\mathsf{c} - 1]_\mathsf{B}$
    $\tau_\mathsf{expected} = \mathsf{frag}[\ell_\mathsf{c} : \ell_\mathsf{c} + \ell_\mathsf{tag} - 1]_\mathsf{B}$
    $\mathsf{frag} = \mathsf{frag}[\ell_\mathsf{c} + \ell_\mathsf{tag} : |\mathsf{frag}|_\mathsf{B} - 1]_\mathsf{B}$
    $\tau \leftarrow \mathsf{Mac}_{\mathsf{k}_m}(c_\mathsf{chunk} \parallel \langle \mathsf{msg\_ctr} \rangle \parallel \langle \mathsf{chunk\_ctr} \rangle)$
    $\mathsf{chunk\_ctr} = \mathsf{chunk\_ctr} + 1$
    if $\mathsf{fail} = \mathsf{true}$
        $m = m \parallel \perp$
    elseif $\tau \neq \tau_\mathsf{expected}$
        $m = m \parallel \perp$
        $\mathsf{fail} = \mathsf{true}$
    else
        $m_\mathsf{chunk} \leftarrow \mathsf{Dec}_{\mathsf{k}_e}(c_\mathsf{chunk})$
        $\mathsf{msg} = \mathsf{msg} \parallel m_\mathsf{chunk}[0 : N - 1]_\mathsf{B}$
        if $m_\mathsf{chunk}[N]_\mathsf{B} \neq \mathtt{0x00}$
            $m = m \parallel \mathsf{msg} \parallel ¶$
            $\mathsf{msg\_ctr} = \mathsf{msg\_ctr} + 1$
            $\mathsf{chunk\_ctr} = 0$
            $\mathsf{msg} = \epsilon$
return $(m, (\mathsf{frag}, \mathsf{msg}, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail}))$
```

Figure 7.1: A byte-oriented version of the original InterMAC scheme, OIM.

scheme OIM $= \left(\overline{\mathsf{Gen}}, \overline{\mathsf{Enc}}, \overline{\mathsf{Dec}}\right)$ *defined in Figure 7.1 gives a symmetric encryption scheme supporting ciphertext fragmentation with associated plaintext space* $\{\mathsf{B}^N\}^*$*.*

The InterMac scheme OIM (O for original) works as follows. First the plaintext is cut into equal size chunks of length N. Each chunk is then encoded by appending a byte b, we call this byte a chunk delimiter, encoding whether the chunk is the last chunk in the plaintext or not. The resulting encoded chunks are then individually encrypted, producing ciphertexts $c^1_\mathsf{chunk}, c^2_\mathsf{chunk}, \ldots$, called ciphertext chunks. A MAC tag is computed over each chunk c^i_chunk, together with the message counter msg_ctr and the chunk counter chunk_ctr, producing a MAC tag τ^i. Finally, all ciphertext chunks and associated MAC tags are concatenated, which yields the final ciphertext

$c = c^1_{\mathsf{chunk}} \parallel \tau^1 \parallel c^2_{\mathsf{chunk}} \parallel \tau^2 \parallel \cdots$. When decrypting, the fragment f is appended to the buffer frag. If the buffer contains more than $\ell_{\mathsf{c}} + \ell_{\mathsf{tag}}$ bytes, the ciphertext chunk c_{chunk} and MAC tag τ_{expected} are extracted, and the extracted data is removed from the buffer frag. The MAC tag is verified over the ciphertext chunk c_{chunk}, message counter $\mathsf{msg\_ctr}$ and chunk counter $\mathsf{chunk\_ctr}$. If the MAC verification fails, the fail flag fail is set and the (error) symbol $\perp$ is appended to the output string buffer m. If the MAC verification passes, the ciphertext chunk is decrypted into a message m_{chunk} and is appended to the current plaintext buffer frag. If the final ciphertext chunk is decrypted the output buffer m is appended with the current plaintext buffer frag and the plaintext delimiter ¶ after which the plaintext buffer is reset.

The message counter and chunk counter are both important to the security of the InterMAC scheme. The former prevents trivial reordering of messages as well as "cross-reordering" where an adversary takes a ciphertext chunk from one ciphertext and substitutes it into another ciphertext. As a consequence, InterMAC does not need to rely on an externally managed sequence counter (or other replay-protection methods) to provide protection against replay attacks. The latter secures against reordering of ciphertext chunks in a ciphertext. These security claims of course depend on the message and chunk counter being authenticated. This is achieved for the original InterMAC scheme because they are included in the MAC scope.

Boldyreva et al. [36] proved that the original InterMAC scheme OIM is $\mathsf{IND\text{-}sfCFA}$, $\mathsf{BH\text{-}sfCFA}$ and $(\ell_{\mathsf{c}} + \ell_{\mathsf{tag}})$-$\mathsf{DOS\text{-}sfCFA}$ secure. They did not prove that OIM is $\mathsf{INT\text{-}sfCTF}$ secure but this can be proven with little effort. The requirements on the internal symmetric encryption scheme SE and the internal message authentication scheme MAC are standard: SE must be $\mathsf{IND\$\text{-}CPA}$ secure, while MAC must be $\mathsf{UF\text{-}CMA}$ secure and the tagging algorithm Mac must be a PRF.

Using InterMAC brings additional overhead on top of any ciphertext expansion introduced by the internal symmetric encryption scheme. Namely, both the chunk encoding and the incremental MACing introduce overhead in the final ciphertext. The precise overhead can be computed as a function of the chunk length N. This function is not necessarily a decreasing function of N because the potential ciphertext expansion of the internal symmetric encryption scheme can increase when increasing the chunk length.[1] We elaborate on this fact in Section 7.7.

7.2.2 Modified InterMAC

The OIM scheme described in Section 7.2.1 is not suitable for use in practical applications because only messages that are a multiple of the chunk length can be encrypted. Fortunately, OIM can be modified to support arbitrary length messages. We extend OIM with padding such that if the message is not a multiple of the chunk length N, we apply padding up to the nearest multiple of N. We use alternating-byte padding. This works by padding with bytes different from the last byte of the message. Specifically, if the last byte of the message is `0x00`, the byte `0x01` is used as the padding byte and if the last byte of the message is not `0x00`, the byte `0x00` is used as the padding byte. This padding scheme is obviously invertible.

In fact, we combine the padding with the chunk delimiter byte to avoid the need to add complete chunks of padding in the event that the plaintext data is already aligned on an N-byte boundary. Specifically, if the message length is already a multiple of the chunk length N, then we set the final chunk delimiter to `0x01`, while if the message length is not a multiple of N, meaning that padding is present, then we set the final chunk delimiter to `0x02`. The final chunk delimiter therefore both indicates when the end of a ciphertext has been reached and whether the final plaintext chunk was padded or not. This combined operation is denoted by $\mathsf{add\_padding}(\cdot,\cdot)$ in Figure 7.2 formally describing the modified InterMAC scheme IM, with output (m, d) denoting the now padded message m and chunk delimiter byte d. Details of the add_padding function are given in Figure 7.3, while the practical effects of padding on bandwidth and speed are explored in detail in Section 7.7.

Note that no padding oracle issues, like those that have plagued TLS's MAC-then-Encrypt construction, will arise during padding removal, because the message and padding will always be protected by an AE scheme in our construction. On the other hand, in a straightforward implementation the running time of the padding removal process (and handling of the final chunk delimiter) would depend on the amount of padding and the value of the chunk delimiter byte, which in turn might lead to leakage of the true message length to an attacker. This is because in such an

[1] Consider, for example, the natural choice of CBC-mode encryption with some padding scheme as the internal encryption scheme SE, with N increasing from below to above a block boundary for the underlying block cipher.

implementation, one would just inspect bytes from right to left in the final chunk until a different byte value was encountered, branching at that point. To avoid this obvious timing side-channel, our $\mathsf{remove\_padding}$ function in Figure 7.3 operates in a constant-time manner. This means that it must operate on every byte of *every* chunk (and not just the last chunk, since we also want to hide the fact that the last chunk is being processed). This has an obvious performance impact compared to using a naive routine for padding removal. We discuss this impact in greater detail in Section 7.7.

In addition to the modifications described above, we make one further change to the original InterMAC scheme OIM in obtaining the modified scheme IM. Instead of performing a two-step process by first encrypting an encoded chunk and then computing a MAC tag over the resulting ciphertext chunk, we use a nonce-based AE scheme. The nonce-based AE scheme is applied directly on the encoded chunks, while the message counter $\mathsf{msg\_ctr}$ and chunk counter $\mathsf{chunk\_ctr}$ are used to generate its nonces. This change means that the chunk counter and message counter are no longer explicitly authenticated. However, their use to construct the nonces means that they are protected by standard security properties of nonce-based AE, as we prove in Section 7.2.3. There are several reasons to make this modification. Firstly, we wish to make the case for using "modern" primitives. Nonce-based AE schemes have seen concrete and systematic analysis, are fast, and are widely supported. Secondly, algorithm agility in InterMAC is easier to achieve when only having to cater for one algorithm type instead of two algorithms that need to be composed; see further discussion in Section 7.3.1. Thirdly, using nonce-based AE makes the presentation of InterMAC cleaner.

The formal definition of IM follows.

Definition 26.
Let $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a nonce-based AE scheme with an associated plaintext space that contains* B^{N+1}*, for some desired* $N \in \mathbb{N}$ *and that has nonce space* $\mathcal{N} = \{0,1\}^n$*. Choose* $a, b \in \mathbb{N}$ *such that* $a + b = n$*. Furthermore, assume that* SE *encrypts all messages of length* $N + 1$ *(counted in bytes) to ciphertexts of length* ℓ_{c} *(counted in bytes). Then the modified InterMAC scheme* $\mathsf{IM} = \left(\overline{\mathsf{Gen}}, \overline{\mathsf{Enc}}, \overline{\mathsf{Dec}}\right)$ *defined in Figure 7.2 gives a symmetric encryption scheme supporting ciphertext fragmentation with associated plaintext space* B^**.*

```
alg. $\overline{\mathsf{Gen}}$
$\mathsf{k} \leftarrow \mathsf{Gen}$
$\mathsf{fail} = \mathsf{false}$
$\mathsf{msg\_ctr} = 0$
$\mathsf{chunk\_ctr} = 0$
$\sigma = \mathsf{msg\_ctr}$
$\varrho = (\epsilon, \epsilon, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail})$
return $(\mathsf{k}, \sigma, \varrho)$
```

```
alg. $\overline{\mathsf{Enc}}_{\mathsf{k}}(m, \sigma)$
$\mathsf{msg\_ctr} \leftarrow \mathsf{Parse}(\sigma)$
$c = \epsilon$
$(m, d) \leftarrow \mathsf{add\_padding}(m, N)$
for $\mathsf{chunk\_ctr} = 0 \ldots \left(\frac{|m|_{\mathsf{B}}}{N} - 1\right)$
  $l = \mathsf{chunk\_ctr} \cdot N$
  $r = (\mathsf{chunk\_ctr} + 1) \cdot N - 1$
  $m_{\mathsf{chunk}} = m[l : r]_{\mathsf{B}}$
  $\mathsf{nonce} = \langle \mathsf{msg\_ctr} \rangle_a \parallel \langle \mathsf{chunk\_ctr} \rangle_b$
  if $q = |m|_{\mathsf{B}}$
    $c_{\mathsf{chunk}} \leftarrow \mathsf{Enc}_{\mathsf{k}}(m_{\mathsf{chunk}} \parallel d, \mathsf{nonce})$
  else
    $c_{\mathsf{chunk}} \leftarrow \mathsf{Enc}_{\mathsf{k}}(m_{\mathsf{chunk}} \parallel \mathtt{0x00},$
                 $\mathsf{nonce})$
  $c = c \parallel c_{\mathsf{chunk}}$
$\mathsf{msg\_ctr} = \mathsf{msg\_ctr} + 1$
return $(c, \mathsf{msg\_ctr})$
```

```
alg. $\overline{\mathsf{Dec}}_{\mathsf{k}}(f, \varrho)$
$(\mathsf{frag}, \mathsf{msg}, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail}) \leftarrow \mathsf{Parse}(\varrho)$
$\mathsf{frag} = \mathsf{frag} \parallel f$
$m = \epsilon$
while $|\mathsf{frag}|_{\mathsf{B}} \geq \ell_{\mathsf{c}}$
  $c_{\mathsf{chunk}} = \mathsf{frag}[0 : \ell_{\mathsf{c}} - 1]_{\mathsf{B}}$
  $\mathsf{frag} = \mathsf{frag}[\ell_{\mathsf{c}} : |\mathsf{frag}|_{\mathsf{B}} - 1]_{\mathsf{B}}$
  $\mathsf{nonce} = \langle \mathsf{msg\_ctr} \rangle_a \parallel \langle \mathsf{chunk\_ctr} \rangle_b$
  $m_{\mathsf{chunk}} \leftarrow \mathsf{Dec}_{\mathsf{k}}(c_{\mathsf{chunk}}, \mathsf{nonce})$
  $\mathsf{chunk\_ctr} = \mathsf{chunk\_ctr} + 1$
  if $\mathsf{fail} = \mathsf{true}$
    $m = m \parallel \bot$
  elseif $m_{\mathsf{chunk}} = \bot$
    $m = m \parallel \bot$
    $\mathsf{fail} = \mathsf{true}$
  else
    $\mathsf{chunk\_del} = m_{\mathsf{chunk}}[N]_{\mathsf{B}}$
    $m_{\mathsf{chunk}} \leftarrow \mathsf{remove\_padding}(m_{\mathsf{chunk}}[0 : N - 1]_{\mathsf{B}},$
                      $N, \mathsf{chunk\_del})$
    if $\mathsf{chunk\_del} \neq \mathtt{0x00}$
      $m = m \parallel \mathsf{msg} \parallel m_{\mathsf{chunk}} \parallel \P$
      $\mathsf{msg\_ctr} = \mathsf{msg\_ctr} + 1$
      $\mathsf{chunk\_ctr} = 0$
      $\mathsf{msg} = \epsilon$
    else
      $\mathsf{msg} = \mathsf{msg} \parallel m_{\mathsf{chunk}}$
return $(m, (\mathsf{frag}, \mathsf{msg}, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail}))$
```

Figure 7.2: The modified InterMAC scheme IM; functions add_padding and remove_padding are defined in Figure 7.3.

The modified InterMAC construction IM exhibits the same security properties as the original InterMAC construction OIM. In the next section, we formally prove that this is indeed true.

7.2.3 Security Analysis of IM

We now turn to the task of formally proving that the changes made to the InterMAC construction OIM in producing IM do not change its security properties. The proofs for IND-sfCFA, BH-sfCFA and DOS-sfCFA security of OIM appeared in [36]. However, the security model turned out to be buggy (as discussed in Chapter 3), and so the proofs for OIM cannot be safely relied upon for IM.

Before diving into the proofs, we start by defining an event **BAD**. In the rest of this

```
alg. add_padding($m, N$)
  size = $|m|_B$
  mod = size mod $N$
  if mod = 0
    return ($m$, 0x01)
  $\ell_{pad} = N -$ mod
  if $m[\text{size} - 1]_B$ = 0x00
    padbyte = 0x01
  else
    padbyte = 0x00
  repeat $\ell_{pad}$ times
    $m = m$ || padbyte
  return ($m$, 0x02)
```

```
alg. remove_padding($m, N$, chunk_del)
  padbyte = $m[N-1]_B$
  $\ell_{pad} = 0$
  flag = 0
  for $i = 0 \ldots N-2$
    flag = flag | ($m[N-i]_B \oplus$ padbyte)
    lsb = (flag | (−flag)) $\gg 7$
    add = lsb $\oplus$ 0x01
    $\ell_{pad} = \ell_{pad}$ + add
  mult = chunk_del · (chunk_del − 1) $\gg 1$
  $\ell_{pad} = \ell_{pad}$ · mult
  $m = m[0, N - \ell_{pad} - 1]_B$
  return $m$
```

Figure 7.3: Functions to add byte-alternating padding and compute chunk delimiter, and to remove byte-alternating padding. The variable flag is to be interpreted as an unsigned 8-bit integer and $-n$ is defined as the operation $2^8 - n$. Beware that some systems/languages/compilers may not respect these conventions.

section, when referring to the encryption oracle, we mean either ENC, LR or LR-BH. Likewise, when referring to the decryption oracle, we mean either DEC or DEC-DOS.

Assume $\mathsf{f}_1, \mathsf{f}_2, \mathsf{f}_3, \ldots$ are fragments queried to the decryption oracle and that the sync flag at some point is set to false. Let v be the unique integer such that before querying f_v, the sync flag is set to true, but after the decryption oracle returns on the query f_v, the sync flag is set to false. Let SF_s denote the (byte) string $\mathsf{f}_1 \parallel \mathsf{f}_2 \parallel \cdots \parallel \mathsf{f}_s$. SF_s is the (in order) concatenation of all (fragment) bytes queried to the decryption oracle after s queries. For $s \geq 1$, let CS_s denote the (byte) string $||(\mathcal{L}_\mathsf{C}[0 : s-1])$, where $\mathcal{L}_\mathsf{C}$ is as defined in Figure 3.7. We let i_e denote the specific value of the variable of the same name in Figure 3.7 at the point in time where f_v is queried to the decryption oracle. Thus, CS_{i_e} is the (in order) concatenation of all (ciphertext) bytes returned by the encryption oracle after i_e encryption queries.

Assume that f_v is queried to the decryption query. There are exactly two sets of conditions on CS and SF for which the sync flag can be set to false during the processing of f_v. The two sets of conditions are:

A. $\underline{|\mathsf{SF}_v|_\mathsf{B} > |\mathsf{CS}_{i_e}|_\mathsf{B} \text{ and } \mathsf{CS}_{i_e} \preceq \mathsf{SF}_v}$. We can find $\lambda \in \mathbb{N} \cup 0$ such that

$$|\mathsf{CS}_{i_e}|_\mathsf{B} \leq \lambda \cdot \ell_\mathsf{c} < (\lambda + 1) \cdot \ell_\mathsf{c} - 1,$$
$$\mathsf{S_F}[\lambda \cdot \ell_\mathsf{c} : (\lambda+1) \cdot \ell_\mathsf{c} - 1] \leftarrow \mathsf{frag}[0 : \ell_\mathsf{c} - 1]_\mathsf{B}.$$

Assume λ is minimal and set $\delta = \mathsf{S_F}[\lambda \cdot \ell_\mathsf{c} : (\lambda + 1) \cdot \ell_\mathsf{c} - 1]$. The assignment in the second line above happens at some point during the execution of the "while" loop in the decryption algorithm of IM and implies that, at some point, δ is an input to Dec.

B. There exists an integer μ such that $\mathsf{S_F}_v[\mu]_\mathsf{B} \neq \mathsf{CS}_{i_e}[\mu]_\mathsf{B}$. Assume μ is minimal. Let $t \leq i_e$ and λ be the unique integers satisfying:

$$|\mathsf{CS}_{t-1}|_\mathsf{B} \leq \lambda \cdot \ell_\mathsf{c} \leq \mu \leq (\lambda + 1) \cdot \ell_\mathsf{c} - 1 < |\mathsf{CS}_t|_\mathsf{B} \, .$$

We must have $j_d = t$ and therefore $\mathsf{CS}_{j_d} \npreceq \mathsf{S_F}_v$. Set $\delta = \mathsf{S_F}[\lambda \cdot \ell_\mathsf{c} : (\lambda + 1) \cdot \ell_\mathsf{c} - 1]$.

In both cases, δ is an input to the decryption algorithm Dec of the underlying nonce-based encryption scheme. Furthermore, the output from the call $\mathsf{Dec_k}(\delta, \mathsf{nonce})$ is recorded in m (since $m \neq \varepsilon$ for in-sync ciphertext). The decryption oracle filters all previous in-sync plaintext decrypted by Dec, which won't be returned to an adversary. Note that we know the exact position at which the output from $\mathsf{Dec_k}(\delta, \mathsf{nonce})$ appears in m without knowledge of the key k, because of the ℓ_c-chunk processing. Denote this position by m_δ. Using δ, we define the following event:

BAD: Dec does not return $\perp$ on input δ.

In all the following proofs, we will make heavy use of the event **BAD**. A subtlety in the theorems below is that their proofs are only valid under the following restrictions on the adversary:

R1 The adversary must make strictly less than 2^a encryption queries and each query must consist of strictly less than $N \cdot 2^b$ bytes.

R2 The adversary must restrict its decryption queries such that the total number of messages decrypted is strictly less than 2^a and each message must consist of strictly less than $N \cdot 2^b$ bytes.

The values a and b refer to the parameters in IM controlling the bit-lengths of the message counter and chunk counter, respectively. Restrictions R1 and R2 are necessary to ensure that the nonce used internally in IM does not repeat. The first theorem shows that IM is ℓ_c-DOS-sfCFA secure.

Theorem 11 (IM is ℓ_{c}-DOS-sfCFA secure)**.**
Let IM *be instantiated with the nonce-based AE scheme* $\mathsf{nSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$*. For any adversary* $\mathcal{A}_{\mathsf{dos}}$*, respecting restrictions* $\mathsf{R1}$ *and* $\mathsf{R2}$*, against* IM*, there exists an* nAE *adversary* $\mathcal{A}_{\mathsf{nae}}$ *against* nSE *such that:*

$$\mathsf{Adv}^{\ell_{\mathsf{c}}\text{-}\mathsf{dos}\text{-}\mathsf{sfcfa}}_{\mathsf{IM}}(\mathcal{A}_{\mathsf{dos}}) \leq \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}_{\mathsf{nae}}). \tag{7.1}$$

If $\mathcal{A}_{\mathsf{dos}}$ *makes* q_e *encryption queries totalling* μ_e *bytes and* q_d *decryption queries totalling* μ_d *bytes, then* $\mathcal{A}_{\mathsf{nae}}$ *makes at most* $\left\lfloor \frac{\mu_e}{N} \right\rfloor + q_e$ *encryption queries totalling at most* $\mu_e + q_e \cdot (N+1)$ *bytes and at most* $\left\lfloor \frac{\mu_d}{\ell_{\mathsf{c}}} \right\rfloor$ *decryption queries totalling* μ_d *bytes.*

Proof. It is always possible for an adversary to win the n-DOS-sfCFA game for $n < \ell_{\mathsf{c}}$ because the IM construction processes fragments in segments of ℓ_{c} bytes; an adversary could bring the decryption oracle out-of-sync and then query with a fragment of size $\ell_{\mathsf{c}} - 1$. When $n \geq \ell_{\mathsf{c}}$, we will use the event **BAD** to upper bound the probability of the adversary winning. If $\mathcal{A}_{\mathsf{dos}}$ succeeds then event **BAD** must have occurred, otherwise IM would only output $\perp$, and the adversary would never win. Therefore:

$$\mathsf{Adv}^{\mathsf{n}\text{-}\mathsf{dos}\text{-}\mathsf{sfcfa}}_{\mathsf{IM}}(\mathcal{A}_{\mathsf{dos}}) \leq \Pr[\mathbf{BAD}]. \tag{7.2}$$

We next show that the probability of the event **BAD** is bounded by the probability of winning the nAE security game. Let $(\mathcal{O}_1, \mathcal{O}_2)$ be oracles such that $(\mathcal{O}_1, \mathcal{O}_2) \in \{(\mathsf{ENC}, \mathsf{DEC}), (\$, \mathsf{Err})\}$ and let the adversary $\mathcal{A}_{\mathsf{nae}}$ have access to both $\mathcal{O}_1$ and $\mathcal{O}_2$. Define $\mathcal{A}_{\mathsf{nae}}$ as follows:

$\mathcal{A}_{\mathsf{nae}}$**:** Run $\mathcal{A}_{\mathsf{dos}}$ answering its queries to ENC and DEC-DOS as specified below. While answering queries, $\mathcal{A}_{\mathsf{nae}}$ maintains: $\mathsf{S_D}$, $\mathcal{L}_{\mathsf{M}}$, $\mathcal{L}_{\mathsf{C}}$ and $\mathsf{S_F}$ from which $\mathcal{A}_{\mathsf{nae}}$ can detect when the sync flag sync is set to false. At this point in time $\mathcal{A}_{\mathsf{nae}}$ terminates and outputs a guess. If event **BAD** happened (which can be discovered by inspecting position m_δ), $\mathcal{A}_{\mathsf{nae}}$ outputs 1, and 0 otherwise.

$\mathsf{ENC}(\cdot)$**:** On input M, simulate ENC and replace the call to Enc with the oracle $\mathcal{O}_1$.

DEC-$\mathsf{DOS}(\cdot)$**:** On input F, simulate DEC-DOS and replace the call to Dec with the oracle $\mathcal{O}_2$.

The simulations of ENC and DEC-DOS are perfect until the point in time where the sync flag sync is set to false. This is also the point in time $\mathcal{A}_{\mathsf{nae}}$ will terminate. We next analyse the probability of $\mathcal{A}_{\mathsf{nae}}$ outputting 1, given the two different sets of oracles.

Given $(\mathcal{O}_1, \mathcal{O}_2) = (\mathsf{ENC}, \mathsf{DEC})$. If **BAD** happens, then it is obvious that:

$$\Pr[\mathbf{BAD}] \leq \Pr\left[\mathsf{INI} \,:\, \mathcal{A}_{\mathsf{nae}}^{\mathsf{ENC}(\cdot,\cdot),\mathsf{DEC}(\cdot,\cdot)} = 1\right]. \tag{7.3}$$

Given $(\mathcal{O}_1, \mathcal{O}_2) = (\$, \mathsf{Err})$. In this case, it is obvious that **BAD** will never happen, because the decryption oracle will always output $\perp$. Hence:

$$\Pr\left[\mathsf{INI} \,:\, \mathcal{A}_{\mathsf{nae}}^{\$(\cdot,\cdot),\mathsf{Err}(\cdot,\cdot)} = 1\right] = 0. \tag{7.4}$$

From (7.3) and (7.4), we have:

$$\Pr[\mathbf{BAD}] \leq \mathsf{Adv}_{\mathsf{nSE}}^{\mathsf{nAE}}(\mathcal{A}_{\mathsf{nae}}). \tag{7.5}$$

Combining (7.2) and (7.5) yields (7.1).

Suppose $\mathcal{A}_{\mathsf{dos}}$ makes q_e encryption queries totalling μ_e bytes and q_d decryption queries totalling μ_d bytes. Then, $\mathcal{A}_{\mathsf{nae}}$ makes at most $\left\lfloor \frac{\mu_e}{N} \right\rfloor + q_e$ encryption queries totalling at most $\mu_e + q_e \cdot (N+1)$ bytes. The term q_e comes from potential padding while the term $q_e \cdot (N+1)$ comes from potential padding and the 1-byte chunk delimiter encoding. Furthermore, $\mathcal{A}_{\mathsf{nae}}$ makes at most $\left\lfloor \frac{\mu_d}{\ell_{\mathsf{c}}} \right\rfloor$ decryption queries totalling μ_d bytes. □

We proceed to show that IM also provides active boundary hiding.

Theorem 12 (IM is $\mathsf{BH\text{-}sfCFA}$ secure)**.**
Let IM *be instantiated with the nonce-based AE scheme* $\mathsf{nSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$*. For any adversary* $\mathcal{A}_{\mathsf{bhcfa}}$*, respecting restrictions* R1 *and* R2*, against* IM*, there exists an* nAE *adversary* $\mathcal{A}_{\mathsf{nae}}$ *against* nSE *such that:*

$$\mathsf{Adv}_{\mathsf{IM}}^{\mathsf{bh\text{-}sfcfa}}(\mathcal{A}_{\mathsf{bhcfa}}) \leq 4 \cdot \mathsf{Adv}_{\mathsf{nSE}}^{\mathsf{nAE}}(\mathcal{A}_{\mathsf{nae}}). \tag{7.6}$$

Suppose $\mathcal{A}_{\mathsf{sfcfa}}$ *makes* q_e *encryption queries totalling* μ_e *bytes and* q_d *decryption queries totalling* μ_d *bytes. Then* $\mathcal{A}_{\mathsf{nae}}$ *makes at most* $\left\lfloor \frac{\mu_e}{N} \right\rfloor + q_e$ *encryption queries totalling at most* $\mu_e + q_e \cdot (N+1)$ *bytes and at most* $\left\lfloor \frac{\mu_d}{\ell_{\mathsf{c}}} \right\rfloor$ *decryption queries totalling* μ_d *bytes.*

Proof. We prove the theorem through a sequence of games. For each of these games, let WIN represent the event that the adversary guesses the bit b correctly.

$\mathsf{G0}$ This is the $\mathsf{BH\text{-}sfCFA}$ game instantiated with IM. Hence:

$$\Pr\left[\mathsf{INI} \,:\, \mathcal{A}_{\mathsf{bhcfa}}^{\mathsf{LR\text{-}BH}(b,\cdot,\cdot),\mathsf{DEC}(\cdot)} = b\right] = \Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{bhcfa}}) \,:\, \mathsf{WIN}]. \tag{7.7}$$

G1 In this game, we modify the decryption oracle. When the sync flag sync is set to false the output buffer m is set to a sequence of $\perp$ symbols. The number of $\perp$ symbols in the sequence is computed as the number of chunks contained in m after the remainder operation between $\mathsf{S_D}$ and (possibly a substring of) $\mathcal{L}_\mathsf{M}$. In this computation, if the symbol $\perp$ appears in m, it counts as a chunk. In addition, some chunks might not consist of N bytes, since padding could have been removed but such occurrences can be detected by looking for the end-of-message symbol ¶. In all subsequent decryption oracle queries, the output buffer m is also replaced by a sequence of $\perp$ symbols. The number of $\perp$ symbols is the number of segments, of length ℓ_c bytes, processed by $\overline{\mathsf{Dec}}$. The games $\mathsf{G0}$ and $\mathsf{G1}$ are identical until to the point in time where the sync flag is set to false. In the case of IM, the games will remain identical if **BAD** does not occur. Hence:

$$\Pr[\mathsf{G0}(\mathcal{A}_\mathsf{bhcfa}) : \mathsf{WIN}] - \Pr[\mathsf{G1}(\mathcal{A}_\mathsf{bhcfa}) : \mathsf{WIN}] \leq \Pr[\mathbf{BAD}]. \tag{7.8}$$

Using the same arguments as in the proof of Theorem 11, we can construct an adversary $\mathcal{A}^1_\mathsf{nae}$ such that:

$$\Pr[\mathbf{BAD}] \leq \mathsf{Adv}^\mathsf{nAE}_\mathsf{nSE}(\mathcal{A}^1_\mathsf{nae}). \tag{7.9}$$

G2 In this game, we modify the decryption oracle to utilise $\mathcal{L}_\mathsf{M}$, $\mathcal{L}_\mathsf{C}$, and $\mathsf{S_F}$ to simulate when the sync flag should be set to false and how long the sequence of $\perp$ symbols should be. This modification does not alter the output of the decryption oracle and therefore, does not change the output distribution. This implies:

$$\Pr[\mathsf{G1}(\mathcal{A}_\mathsf{bhcfa}) : \mathsf{WIN}] = \Pr[\mathsf{G2}(\mathcal{A}_\mathsf{bhcfa}) : \mathsf{WIN}]. \tag{7.10}$$

G3 In this game, we replace calls to Enc with calls to $\$$ and replace calls to Dec with calls to Err. Let $\mathcal{A}^2_\mathsf{nae}$ be an adversary having access to oracles $(\mathcal{O}_1, \mathcal{O}_2) \in \{(\mathsf{ENC}, \mathsf{DEC}), (\$, \mathsf{Err})\}$. Define $\mathcal{A}^2_\mathsf{nae}$ as follows: $\mathcal{A}^2_\mathsf{nae}$ runs $\mathcal{A}_\mathsf{bhcfa}$ using oracles $(\mathcal{O}_1, \mathcal{O}_2)$ to simulate the encryption and decryption oracles. If $\mathcal{A}_\mathsf{bhcfa}$ wins, $\mathcal{A}^2_\mathsf{nae}$ outputs 1, otherwise outputs 0. Now, if $(\mathcal{O}_1, \mathcal{O}_2) = (\mathsf{ENC}, \mathsf{DEC})$, $\mathcal{A}^2_\mathsf{nae}$ perfectly simulates Game $\mathsf{G2}$ and if WIN occurs, $\mathcal{A}^2_\mathsf{nae}$ outputs 1. If $(\mathcal{O}_1, \mathcal{O}_2) = (\$, \mathsf{Err})$, $\mathcal{A}^2_\mathsf{nae}$ perfectly simulates Game $\mathsf{G3}$ and if WIN occurs, $\mathcal{A}^2_\mathsf{nae}$ outputs 1. Therefore:

$$\begin{aligned}
&\Pr[\mathsf{G2}(\mathcal{A}_\mathsf{bhcfa}) : \mathsf{WIN}] - \Pr[\mathsf{G3}(\mathcal{A}_\mathsf{bhcfa}) : \mathsf{WIN}] \\
&= \Pr\left[\mathsf{INI} : \mathcal{A}^2_\mathsf{nae}{}^{\mathsf{ENC}(\cdot,\cdot),\mathsf{DEC}(\cdot,\cdot)} = 1\right] - \Pr\left[\mathsf{INI} : \mathcal{A}^2_\mathsf{nae}{}^{\$(\cdot,\cdot),\mathsf{Err}(\cdot,\cdot)} = 1\right] \\
&\leq \mathsf{Adv}^\mathsf{nAE}_\mathsf{nSE}(\mathcal{A}^2_\mathsf{nae}).
\end{aligned} \tag{7.11}$$

Consider Game $\mathsf{G3}$. On decryption queries, the adversary can only obtain strings containing the symbol $\perp$. On encryption queries, the adversary only obtains byte strings consisting of concatenations of uniformly random strings returned by the oracle $\$$. The adversary, therefore, does not learn anything about the bit b from the combination of decryption and encryption queries. Hence:

$$\Pr[\mathsf{G3}(\mathcal{A}_{\mathsf{bhcfa}}) : \mathsf{WIN}] = \frac{1}{2}. \tag{7.12}$$

Set $\mathcal{A}_{\mathsf{nae}}$ to be either $\mathcal{A}^1_{\mathsf{nae}}$ or $\mathcal{A}^2_{\mathsf{nae}}$ such that $\mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}_{\mathsf{nae}})$ is maximised. Combining (7.7), (7.8), (7.9), (7.10), (7.11), and (7.12) yields (7.6).

Suppose $\mathcal{A}_{\mathsf{sfcfa}}$ makes q_e encryption queries totalling μ_e bytes and q_d decryption queries totalling μ_d bytes. Then, $\mathcal{A}_{\mathsf{nae}}$ makes at most $\left\lfloor \frac{\mu_e}{N} \right\rfloor + q_e$ encryption queries totalling at most $\mu_e + q_e \cdot (N+1)$ bytes. The term q_e comes from potential padding while the term $q_e \cdot (N+1)$ comes from potential padding and the 1-byte chunk delimiter encoding. Furthermore, $\mathcal{A}_{\mathsf{nae}}$ makes at most $\left\lfloor \frac{\mu_d}{\ell_{\mathsf{c}}} \right\rfloor$ decryption queries totalling μ_d bytes. Note, $\mathcal{A}_{\mathsf{nae}^1}$ and $\mathcal{A}^2_{\mathsf{nae}}$ consume the same number of resources. □

In Game $\mathsf{G2}$, we could remove the decryption algorithm, since it is not needed to simulate the correct output, and get a tighter bound than the one presented in (7.6) with a more efficient adversary. However, it helps the understanding in Game $\mathsf{G3}$ when building the second nAE adversary. Combining Theorem 12 and Theorem 3, we can conclude that IM is also $\mathsf{IND\text{-}sfCFA}$ secure (under restrictions $\mathsf{R1}$ and $\mathsf{R2}$).

Finally, we prove that IM is $\mathsf{INT\text{-}sfCTF}$ secure. Because this definition is new the original InterMAC scheme OIM was not proved $\mathsf{INT\text{-}sfCTF}$ secure in [36]. However, only minor modifications in the proof below are required to also prove that OIM is $\mathsf{INT\text{-}sfCTF}$ secure.

Theorem 13 (IM is $\mathsf{INT\text{-}sfCTF}$ secure)**.**
Let IM *be instantiated with the nonce-based AE scheme* $\mathsf{nSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$*. For any adversary* $\mathcal{A}_{\mathsf{ctf}}$*, respecting restrictions* $\mathsf{R1}$ *and* $\mathsf{R2}$*, against* IM*, there exists an* nAE *adversary* $\mathcal{A}_{\mathsf{nae}}$ *against* nSE *such that:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}ctf}}_{\mathsf{IM}}(\mathcal{A}_{\mathsf{ctf}}) \le \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}_{\mathsf{nae}}). \tag{7.13}$$

Suppose $\mathcal{A}_{\mathsf{ctf}}$ *makes* q_e *encryption queries totalling* μ_e *bytes and* q_d *decryption queries* *totalling μ_d bytes. Then $\mathcal{A}_{\mathsf{nae}}$ makes at most $\left\lfloor \frac{\mu_e}{N} \right\rfloor + q_e$ encryption queries totalling at most $\mu_e + q_e \cdot (N+1)$ bytes and at most $\left\lfloor \frac{\mu_d}{\ell_{\mathsf{c}}} \right\rfloor$ decryption queries totalling μ_d bytes.*

Proof. We prove the theorem through a sequence of games.

G0 This is the INT-sfCTF game instantiated with IM. Hence:

$$\Pr\left[\mathsf{INI},\ \mathcal{A}_{\mathsf{ctf}}^{\mathsf{ENC}(\cdot),\mathsf{DEC}(\cdot)} \ :\ \mathsf{FORGE}\right] = \Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}]\,. \tag{7.14}$$

G1 In this game, we make modifications identical to the changes made in Game G1 in the proof of Theorem 12. Using an identical argument, we can construct an adversary $\mathcal{A}_{\mathsf{nae}}$ such that:

$$\Pr[\mathsf{G0}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}] - \Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}] \leq \Pr[\mathbf{BAD}] \leq \mathsf{Adv}_{\mathsf{nSE}}^{\mathsf{nAE}}(\mathcal{A}_{\mathsf{nae}}). \tag{7.15}$$

Observe that the decryption oracle in Game G1 will never output anything from the set $\{0, 1, ¶\}^+$. Hence, the event FORGE will never occur in Game G1. Therefore:

$$\Pr[\mathsf{G1}(\mathcal{A}_{\mathsf{ctf}})\ :\ \mathsf{FORGE}] = 0. \tag{7.16}$$

Combining (7.14), (7.15), and (7.16) yields (7.13).

Suppose $\mathcal{A}_{\mathsf{sfcfa}}$ makes q_e encryption queries totalling μ_e bytes and q_d decryption queries totalling μ_d bytes. Then, $\mathcal{A}_{\mathsf{nae}}$ makes at most $\left\lfloor \frac{\mu_e}{N} \right\rfloor + q_e$ encryption queries totalling at most $\mu_e + q_e \cdot (N+1)$ bytes. The term q_e comes from potential padding while the term $q_e \cdot (N+1)$ comes from potential padding and the 1-byte chunk delimiter encoding. Furthermore, $\mathcal{A}_{\mathsf{nae}}$ makes at most $\left\lfloor \frac{\mu_d}{\ell_{\mathsf{c}}} \right\rfloor$ decryption queries totalling μ_d bytes. □

This concludes our formal security analysis of IM.

7.3 libInterMAC

This section describes libInterMAC, a reference C-implementation of the IM scheme presented in Section 7.2.2. We also touch on some of the challenges involved in bringing InterMAC into practice. Especially, we discuss side-channels and possible mitigations, cf. Section 7.6. The code of libInterMAC is available at [3].

libInterMAC has not seen a security review by professional code reviewers. However, a best-effort has been made to implement IM soundly and faithfully. A number of unit tests exists in the code base, that verify a the functional correctness the libInterMAC APIs.

7.3.1 Design Principles

The design of libInterMAC is guided by the following two design principles: *ease of use* and *extensibility*. Furthermore, much effort has been made to carry over proven theoretical security properties of IM to the implementation.

Ease of Use

Cryptographic software often suffers from poor usability and large APIs. Coupled with the many pitfalls involved when implementing and using cryptography, poor usability can have devastating consequences [31, 2]. Consider, for example, software that implements the nonce-based AE scheme AES-GCM and allows the user to specify the nonce for each invocation of the encryption algorithm. If the user at any point reuses a nonce, under the same encryption key, part of the key is leaked [89, 35].

We have attempted to minimise the libInterMAC API as well as making it intuitive to use. The API consists of only three functions, `im_init()`, `im_encrypt()` and `im_decrypt()`, and it is easy to distinguish the functionality of each function simply by its name.

Extensibility

libInterMAC defines an interface to represent the nonce-based AE scheme required in the IM construction and utilises the interface internally. Any nonce-based AE scheme with the same interface can, therefore, be used, allowing users to extend libInterMAC with other nonce-based AE scheme implementations. In other words, libInterMAC supports cryptographic algorithm agility, with respect to the internal nonce-based AE scheme, and aims to adhere to [83] where possible. Cryptographic algorithm agility is vital to facilitate a quick transition away from schemes that have become insecure or that have been made otherwise obsolete.

7.3.2 State Management

The libInterMAC state consists of various elements including the current message and chunk counters, the user-chosen key, the chunk length and the internal nonce-based AE scheme. There is no direct method (i.e. through a public API or de-referencing state fields) by which a user can modify the state. Only when initialising libInterMAC, through the public initialisation function `im_init()`, do user-supplied parameters affect the state. However, these parameters are only used to initialise the state and are all sanitised. Making state management opaque enhances protection against unintentional user-triggered state corruption, reuse of counters, etc. Naturally, although the internal AE schemes used in libInterMAC consume nonces, there is no interface by which the user can modify those nonces.

7.3.3 Internal Nonce Construction

libInterMAC generates and updates the nonce used in the internal AE scheme according to the following procedure: nonces are generated as described in the definition of IM in Figure 7.2 with $a = 64$ and $b = 32$. That is, the first part of the nonce is the 64 LSBs of the message counter msg_ctr, and the last part of the nonce is the 32 LSBs of the chunk counter chunk_ctr, producing a 96-bit nonce. Recall that for each processed message the message counter is incremented and for each processed chunk the chunk counter is incremented. The nonce-based AE scheme encryption and decryption algorithms are applied to each chunk. Therefore, the nonce will not repeat before 2^{64} messages have been processed or divided into more than 2^{32} chunks. Consequently, for a fixed key, libInterMAC can encrypt a maximum of 2^{64} messages each up to a maximum of $N \cdot 2^{32}$ bytes before nonce repetition. Note, however, that the key usage limit also depends on the chosen internal nonce-based AE scheme which may impose further restrictions. Section 7.4 discuss specific restrictions for each supported nonce-based AE scheme.

7.4 Supported Nonce-based Symmetric Encryption Schemes

We have implemented support for two different nonce-based AE schemes: ChaCha20-Poly1305 and AES-GCM. Below we describe how each scheme is implemented, and how they consume the nonce generated in libInterMAC.

```
alg. ChaCha20-Poly1305-enc(k, nonce, $m$)
block_ctr = $0$
$\mathsf{k_{poly}} \leftarrow$ ChaCha20(k, 0x00$^{32}$, nonce, block_ctr)
block_ctr = $1$
$\tau \leftarrow$ Poly1305MAC($\mathsf{k_{poly}}$, $c$)
$c \leftarrow$ ChaCha20(k, $m$, nonce, block_ctr)
$\tau \leftarrow$ Poly1305MAC($\mathsf{k_{poly}}$, $c$)
return $c \parallel \tau$

alg. ChaCha20-Poly1305-dec(k, nonce, $c$, $\tau_{\mathsf{expected}}$)
block_ctr = $0$
$\mathsf{k_{poly}} \leftarrow$ ChaCha20(k, 0x00$^{32}$, nonce, block_ctr)
$\tau \leftarrow$ Poly1305MAC($\mathsf{k_{poly}}$, $c$)
if $\tau \neq \tau_{\mathsf{expected}}$
  return $\perp$
block_ctr = $1$
$m \leftarrow$ ChaCha20(k, $c$, nonce, block_ctr)
return $m$
```

Figure 7.4: The nonce-based AE scheme ChaCha20-Poly1305.

7.4.1 ChaCha20-Poly1305

The nonce-based AE scheme ChaCha20-Poly1305 implemented in libInterMAC closely follows the AEAD composition of ChaCha20 and Poly1305MAC defined in RFC 7539 [113] and is depicted in Figure 7.4, following the notation defined in Section 6.3. For convenience, the pseudo-code presents the encryption and decryption operations separately. ChaCha20-Poly1305 deviates slightly from what is described in RFC 7539. First, we dispense with the AD (additional data), for the simple reason that it is not needed. If the message counter and chunk counter were not used to generate the nonce, they could have been added as additional data. Second, we exclude the padding and length fields in the input to the Poly1305MAC algorithm. In general, these fields are important to the security of the construction. For example, without the length fields an adversary can divide up the received aad $\parallel c \parallel \tau$ in a different way than the sender (e.g. it can make the last part of the additional data aad be the first part of the ciphertext c) allowing trivial forgery attacks. However, these fields can be left out under certain conditions. One such instance is when additional data is not present and when the plaintext/ciphertext is processed in fixed lengths. Both of these conditions hold true in IM.

In Figure 7.4, the string `0x00`^{32} consists of 32 `0x00`-bytes. When used as input in the ChaCha20 stream cipher the resulting output is the first 32 bytes of the ChaCha20 block function. Note that we must increment the block counter block_ctr between the two invocations of ChaCha20.

7.4.2 AES-GCM

The AES-GCM nonce-based AE scheme is specified in [107]. In libInterMAC, the AES-GCM scheme is implemented using the Libcrypto EVP API with 128-bit AES. Libcrypto is part of the OpenSSL Toolkit [46] and the EVP functions provide a high-level interface to cryptographic functions in Libcrypto. We opted to implement AES-GCM using Libcrypto's EVP API because it is widely supported and it automatically applies hardware acceleration when available. This, in turn, can dramatically increase the performance of AES-GCM.

7.4.3 Why ChaCha20-Poly1305 and AES-GCM?

The decision to support ChaCha20-Poly1305 and AES-GCM as nonce-based AE schemes is based on the individual strength of each scheme and the diversity they provide.

The encryption parts of AES-GCM and ChaCha20-Poly1305 are based on two different design ideas. If one design is found to be weak, then the user can switch to the other scheme. Because of design diversity it is unlikely that the other scheme would possess the same weakness. Furthermore, ChaCha20-Poly1305 is designed to be fast on general purpose CPUs without dedicated cryptography instructions, e.g. mobile phones. AES-GCM can make use of dedicated CPU instruction sets (`aes-ni` and `pclmulqdq`) that drastically increases its performance.

A positive feature of AES-GCM and ChaCha20-Poly1305 is that they do not have any ciphertext expansion (beyond the MAC tag) in the case where the nonce does not need to be sent on the wire. This makes them attractive to use in the IM scheme. The reason is that IM essentially performs several (shorter) encryptions on each message. If the underlying nonce-based AE scheme had a large ciphertext expansion, it would lead to an amplified ciphertext expansion in the IM scheme.

A performance comparison for IM using AES-GCM and ChaCha20-Poly1305 can

Nonce-based AE scheme	Chunk length	libInterMAC chunk length limit
AES-GCM	$< 2^{36} - 2^5$	2^{32}
ChaCha20-Poly1305	$< 2^{38}$	2^{32}

Table 7.1: The middle column contains derived chunk length restrictions for the internal nonce-based AE schemes implemented in libInterMAC. The right-most column shows the limit on the size of the chunk length implemented in libInterMAC. All lengths are counted in bytes.

be found in Section 7.7. A comparison for IM when used with AES-GCM and ChaCha20-Poly1305 to implement SSH encryption schemes can be found in Section 7.9.

7.5 libInterMAC Data Limits

When deriving data limits for libInterMAC, our starting point is the security proofs of IM. They show that the underlying nonce-based AE scheme is the dominating factor, and, informally, the only factor to consider when determining data limits for libInterMAC. We, therefore, focus solely on the supported nonce-based AE scheme. Furthermore, instead of deriving explicit data limits, we derive restrictions on a number of parameters used as input to the nonce-based AE schemes, implicitly capturing libInterMAC data limits. We first provide a brief overview of which restrictions libInterMAC implements.

Table 7.1 contains a summary of chunk length restrictions for each supported nonce-based AE scheme. libInterMAC adopts the conservative choice of restricting the chunk length to (strictly less than) 2^{32} for both nonce-based AE schemes. This size seems sufficient and can be natively supported on many platforms. Table 7.2 (top) contains a summary of the restrictions on the number of encrypted chunks as a function of the chunk length when AES-GCM is used as the nonce-based AE scheme. In this case, libInterMAC assumes a maximum attack probability of approximately 2^{-50}. Since the attack probability increases with the number of encryption algorithm invocations, a limit on the allowed number of encrypted chunks when using AES-GCM, is also enforced, see Table 7.2 (bottom). There are no restrictions on the number of encrypted chunks for ChaCha20-Poly1305. Derivations follow in Section 7.5.1 and Section 7.5.2.

chunk length **success prob.**	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^k
2^{-60}	$2^{31.5}$	$2^{30.5}$	$2^{29.5}$	$2^{28.5}$	$2^{27.5}$	$2^{26.5}$	$2^{25.5}$	$2^{24.5}$	$2^{38.5-k}$
2^{-50}	$2^{41.5}$	$2^{40.5}$	$2^{39.5}$	$2^{38.5}$	$2^{37.5}$	$2^{36.5}$	$2^{35.5}$	$2^{34.5}$	$2^{48.5-k}$
2^{-40}	$2^{51.5}$	$2^{50.5}$	$2^{49.5}$	$2^{48.5}$	$2^{47.5}$	$2^{46.5}$	$2^{45.5}$	$2^{44.5}$	$2^{58.5-k}$
libInterMAC **limit on # chunks**	2^{41}	2^{40}	2^{39}	2^{38}	2^{37}	2^{36}	2^{35}	2^{34}	2^{48-k}

Table 7.2: Derived restrictions on the number of encrypted chunks as a function of the attack success probability and chunk length for AES-GCM. The right-most column shows the general formula for computing the restrictions on the number of encryption chunks for different attack success probabilities. The bottom row shows the limits on the number of encrypted chunks for different chunk lengths as implemented in libInterMAC.

7.5.1 ChaCha20-Poly1305 Data Limit Analysis

ChaCha20 must not be used to encrypt more than 2^{38} bytes under the same key-nonce pair $(\mathsf{k}, \mathsf{nonce})$ because the block counter in the ChaCha20 block function is 4 bytes long and ChaCha20 encrypts in 64-byte blocks. Since the nonce is incremented for each processed chunk, the limit will only be reached if the chunk length is extremely large: $N + 1 \geq 2^{38}$.

The success probability of a forgery when using Poly1305MAC increases as the plaintext size grows. Using the language of IM, this implies that the success probability of a forgery increases as the chunk length grows. If we choose the chunk length as $N = 2^k$, forged messages are rejected with a probability close to $1 - v \cdot (2^k + 1)/(2^{106})$ even after authenticating 2^{64} legitimate chunks and v forgery attempts [26].

[117] shows that that security degradation derived for ChaCha20 and Poly1305MAC extends to the ChaCha20-Poly1305 construction. Because of the ChaCha20 estimate, we require in libInterMAC that $N + 1 \leq 2^{38}$ when ChaCha20-Poly1305 is chosen as the internal nonce-based AE scheme. The Poly1305MAC estimate does not impose any restrictions because the key-nonce input to ChaCha20-Poly1305 in libInterMAC changes for each new message, and each message can consist of at most 2^{32} chunks because of the nonce construction.

7.5.2 AES-GCM Data Limit Analysis

Deriving encryption limits for AES-GCM is somewhat involved. Below we give an account of the relevant limits for AES-GCM as used in libInterMAC. AES-GCM can encrypt at most $2^{32} - 2$ blocks of 2^4 bytes per AES-GCM key-nonce pair. This limitation originates from the design of the AES-GCM encryption mode; the internal block counter is 32-bits wide but is initialised to 1 and, in addition, one block is used in conjunction with the output from GHASH to produce the MAC tag. As an effect, the chunk length must be strictly smaller than $2^{36} - 2^5$ bytes. Similar to ChaCha20-Poly1305, the authentication security of AES-GCM degrades as the chunk length increases. Specifically, if the chunk length N is 2^k blocks long (where one AES-GCM block is 16 bytes wide), then a forgery attempt is rejected with probability $1 - 2^{128-k}$ [59].

From [104], we can extract further security degradation estimates for AES-GCM. Firstly, we obtain an upper bound on the number of encryption invocations of $2^{(137-u)/2-k}$ per key, when the block size is 128 bits, the attack success probability for a chosen plaintext distinguishing attacker is 2^{-u}, and the chunk length is $N = 2^k$. Table 7.2 gives an overview of upper bounds for various choices of attack success probability and chunk length. Secondly, we can extract a forgery bound of $2v \cdot (2^{k-4} + 2)/2^{128}$, where v is the number of authentication attempts and the chunk length is $N = 2^k$. If the number of verification attempts is less than 2^{60}, then forged messages are still rejected with a probability close to $1 - 2^{-49}$.

When AES-GCM is chosen as the internal nonce-based AE scheme in libInterMAC, we impose the chunk length restriction and the number of encrypted chunks restriction presented in Table 7.1 and Table 7.2, respectively.

7.6 Side-Channels

In this section, we discuss side-channel attack vectors for IM and libInterMAC, with a particular focus on the extent to which the passive and active boundary hiding notions (which IM is proven to achieve) can be undermined by timing side-channels.

Removing such side-channels in our implementation libInterMAC turns out to be a considerable challenge. The primary complicating factor is our desire to achieve active boundary hiding when also considering side-channel attacks. This goes further than what the formal boundary hiding notion promises, because the formal model does not consider side-channels, but only information that becomes visible to the adversary via output from its encryption and decryption oracles (yet, recall that even this information is sufficient to break boundary hiding for currently supported SSH encryption schemes in OpenSSH, see Chapter 6). When also considering side-channels as a possible attack vector, it becomes paramount that the execution time of the decryption algorithm is independent of ciphertext boundaries; otherwise such information could make it possible for an adversary to infer them.

We present the methods that libInterMAC uses to limit the scope for mounting such side-channel attacks. However, we acknowledge that these methods are not a complete solution. In particular, we do not achieve a full constant-time implementation. In general, such an implementation would anyway require a close co-operation between libInterMAC and any application-layer protocol implementation making use of libInterMAC. This is similar to the situation that exists for the TLS 1.3 Record Protocol, as pointed out in [119, Appendix E.3]. Furthermore, one should also consider the analysis in Section 3.9 if deploying InterMAC: recall, a "reactive" application cannot satisfy active boundary hiding because responses from the application delineate ciphertext boundaries.

7.6.1 Constant-time Padding Removal

Recall that IM uses alternating byte padding in the last chunk of a message to bring it up to the required chunk size. This padding is easy to remove during decryption, simply by inspecting the bytes from right to left and removing them one-by-one until a new byte value is encountered. (There is also a special case in IM because of our use of a distinct value for the chunk delimiter byte when no padding was needed.) However, this is not a constant-time approach, and a timing attack could glean information about the lengths of messages by observing the execution time of padding removal or of the complete decryption operation. As noted above, the TLS 1.3 specification accepts the presence of a similar leakage in its record fragment padding removal mechanism, choosing not to defend against it [119, Appendix E.3], instead stating that: *in general, it is not known how to remove all of these channels because even a constant-time padding removal function will likely feed the content into data-dependent functions.*

Yet to ignore this side-channel when we have targeted boundary hiding security notions seems misguided, even if an attack based on this side-channel would not violate the formal security definitions. For this reason, our padding removal code in Figure 7.3 removes padding (and the chunk delimiter byte) in a constant-time manner. This approach is applied to every chunk, whether it is the final chunk in a message or not, in an effort to hide this information. However, as is evident from the pseudo-code for IM in Figure 7.2, our overall decryption processing is not constant-time. The main issue is the branch on the chunk delimiter byte when deciding whether to perform message finalisation at line 19. Moreover, with our pseudo-code for IM as written in Figure 7.2, a series of ciphertext fragments containing many ciphertexts would take longer to process than a series of ciphertext fragments of the same total length but containing only one ciphertext. This is because there would be a greater number of message finalisation steps (lines 20-23 in Figure 7.2) in the latter case. Of course, one could try to go further to ensure that the finalisation step is also done in constant time on a per-chunk basis, making it independent of the number of messages. We did not pursue this enhancement in libInterMAC, instead stopping at the implementation of constant-time padding removal. The cost of adopting constant-time padding removal is discussed and contrasted against the performance of libInterMAC without constant-time padding removal in Section 7.7.

We have focused on padding removal in part because it is straightforward to make addition of padding during encryption constant time. More importantly, though, we believe that constant-timeness is less critical for encryption than for decryption because a network attacker would not necessarily have a means of measuring encryption times, whereas a network attacker can measure decryption times via timing of error messages, say. Of course, this situation would change when considering a local attacker (with the ability to perform cache timing attacks, for example). In summary, our focus was on addressing the most obvious questions about the potential for constant-time implementations of IM and we acknowledge that our implementation does not achieve constant-timeness throughout. Indeed, eliminating all side-channel leakage that might permit undermining of the boundary-hiding security goal in practice is a challenging future topic of research.

7.6.2 Memory Allocation for InterMAC Decryption

Another situation where a timing-channel could arise in libInterMAC is in the implementation of the decryption buffer that stores the decrypted ciphertext. Recall that when decrypting there is *a priori* no information revealing how long a ciphertext is. This information is first learned when decrypting the last ciphertext chunk in a ciphertext and inspecting the chunk delimiter byte. The decryption function must, therefore, use a buffer that is large enough to store *all* the decrypted ciphertext chunks until the entire ciphertext has arrived *without* knowing the final length of the ciphertext. Below are various strategies for implementing such a buffer in C:

1. Start from an initial decryption buffer of some size s and expand the buffer if necessary using an exponential smoothing approach. That is, if the buffer runs out of memory, re-allocate the buffer (e.g. using the C-function `realloc()`) to a total size of $2 \cdot s$. If the buffer runs out a memory again, expand the buffer again to a total size of $2^2 \cdot s$, i.e. every time the buffers runs out of memory, double the current available memory.
2. Same as (1) but only expand the buffer for a single decrypted chunk at a time. That is, initially $s = |\text{decrypted chunk}|$ and if the buffer runs out of memory expand to a total size of $s + |\text{decrypted chunk}|$.
3. Implement a buffer as a linked list of small buffers. Each buffer will have the size of |decrypted chunk| that are linked as in a linked list.
4. Use a fixed-sized decryption buffer.

Strategy (1) is likely to be the most efficient strategy, balancing the memory requirement with the processing time needed to expand the buffer. However, the time it takes to expand the available memory is dependent on the amount of memory copied and the increase in memory size (at least when assuming use of `realloc()`). This gives a potential for timing leakage, since the decryption time would depend on the ciphertext boundaries.

Strategy (2) is similar to (1) and suffers from the same timing issue. In addition, (2) requires a large number of memory expansions for large ciphertexts, which could impact performance negatively.

Strategy (3) does not directly leak timing information through memory expansions because the expansions are always of the same size and occurs in the same pattern for all ciphertext sizes. However, many applications do not natively support such data structures and is it likely that the decrypted ciphertext must be copied to a different data structure to be further processed in the application anyway. This both decreases performance and can potentially leak timing information. The large number of memory expansions needed can also negatively affect performance as in the case of (2).

Strategy (4) does not have the problems arising in (1), (2) and (3). On the other hand, (1), (2) and (3) all essentially support arbitrary length ciphertexts while (4) does not. Moreover, using a fixed-size buffer imposes restrictions on the use of libInterMAC that might prevent applications from making use of the library.

libInterMAC implements strategy 4, mainly because it is the simplest. In addition, many applications, such as SSH, have a soft packet size limit that restricts ciphertext sizes, making the flexibility offered by the other strategies less important. libInterMAC can be custom-built to set the size of the decryption buffer to the desired size and align with any further restrictions.

7.7 Performance Evaluation

The performance of libInterMAC primarily depends on the choice of the internal nonce-based AE scheme and the choice of the chunk length; a faster scheme will also result in better performance for libInterMAC.

The chunk length plays a more subtle role in the performance of libInterMAC. Recall from Section 7.2 that if the chunk length is N, the message is split into chunks of size N and a single byte is appended to each chunk before the encryption step using the internal nonce-based AE scheme. The number of AES/ChaCha20 operations needed to encrypt an L-byte message for a chunk length N is then equal to $\lceil L/N \rceil \cdot \lceil (N+1)/B \rceil$, where $B = 16$ for AES and $B = 64$ for ChaCha20. To minimise the number of operations, it would seem beneficial to set $N = B - 1 \bmod B$, so that the term $\lceil (N+1)/B \rceil$ does not involve rounding up; on the other hand, this would imply a smaller N, increasing the size of the term $\lceil L/N \rceil$. It is therefore not immediately obvious what the best choice of N is, given B and a specific message length L.

To illustrate the behaviour of the above formula, we assume that AES-GCM is used as the internal nonce-based AE encryption scheme. We focus on comparing the

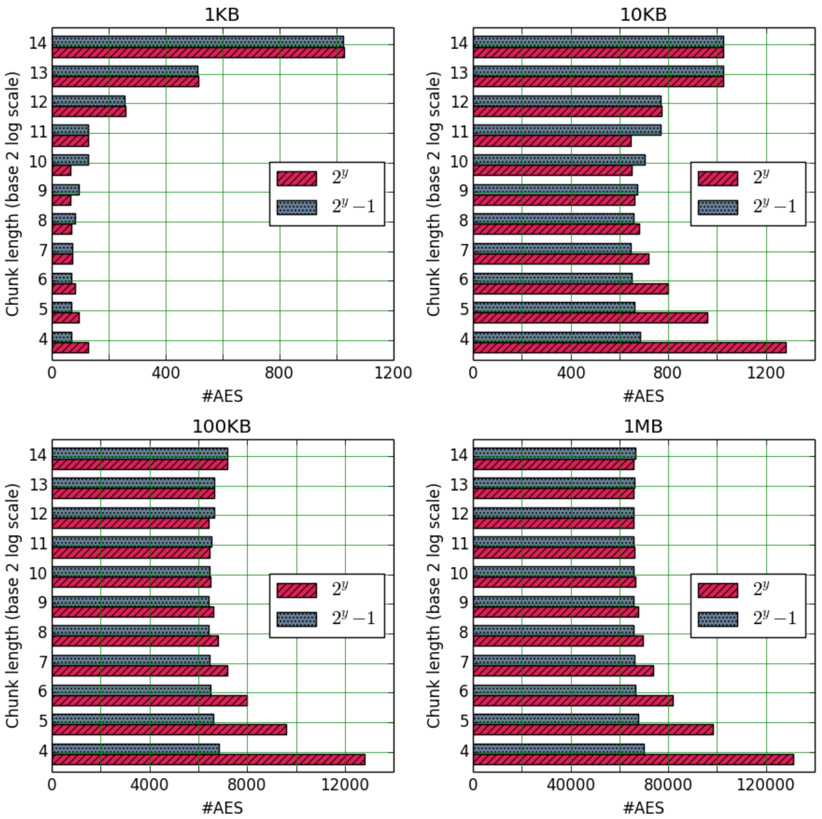

Figure 7.5: Number of AES operations needed to IM-encrypt a message as a function of the chunk length, with the choice of AES-GCM as internal nonce-based AE scheme. Plots are given for 4 different message lengths: 1KB, 10KB, 100KB and 1MB.

behaviour for two sets of chunk lengths: one set that aligns on the block size boundary and one set that aligns on the block size boundary after the addition of the chunk delimiter. First we plot the number of AES operations needed as a function of the chunk length, where the chunk length is a power of two or a power of two minus 1, see Figure 7.5. We also plot the number of AES operations needed as a function of the message length for chunk lengths $2^8 - 1$, 2^8, $2^9 - 1$, 2^9, $2^{10} - 1$, 2^{10}, $2^{11} - 1$ and 2^{11}, see Figure 7.6.

Choosing the chunk length to be $N = 0 \bmod 16$ (red bars in charts) implies that an extra AES computation must be performed for each chunk because the extra byte appended to the chunk pushes the input to the encryption step past the 16-byte block size boundary. When choosing the chunk length to be $N = 15 \bmod 16$ (blue bars in charts) the input to the encryption step aligns with the 16-byte block size, but there will be more chunks to process for long messages. The effect on the number

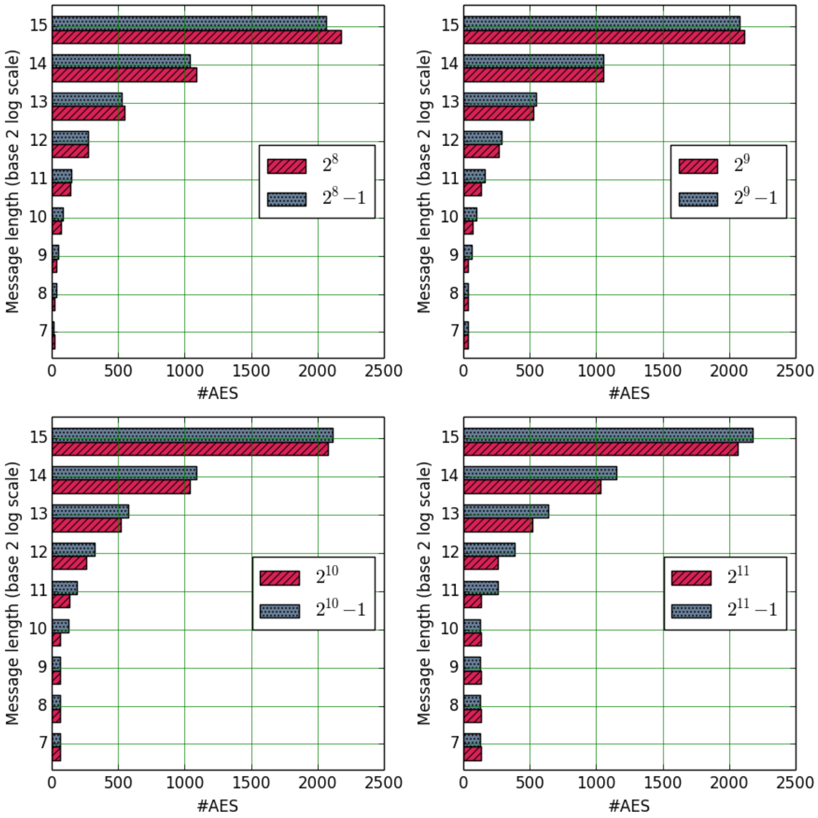

Figure 7.6: Number of AES operations needed to IM-encrypt a message as a function of the message length, with the choice of AES-GCM as internal nonce-based AE scheme. Plots are given for 8 different chunk lengths: $2^8 - 1$, 2^8, $2^9 - 1$, 2^9, $2^{10} - 1$, 2^{10}, $2^{11} - 1$ and 2^{11}.

of AES computations is particularly visible when the chunk length is much smaller than the message length, see charts for message lengths 10KB, 100KB and 1MB in Figure 7.5. As the chunk length grows and the ratio between the message length and the chunk length becomes smaller, the cost between choosing $N = 0 \bmod 16$ and $N = 15 \bmod 16$ evens out and, eventually, shifts in favour of the former.

Figure 7.6 indicates that choosing the chunk length $N = 0 \bmod 16$ is better when the chunk length is close to the message length and that the cross-over (the point at which $N = 15 \bmod 16$ becomes the better choice) happens when the message length is significantly larger than the chunk length. These observations agree with the observations from Figure 7.5.

From a security perspective choosing the chunk length close to the message length decreases DoS resistance. By taking a small performance hit, in terms of the number of AES operations, it is possible to significantly lower the chunk length and thereby increasing the DoS resistance. However, there are other costs associated with decreasing the chunk length. We will further investigate these costs below and in Section 7.9.

To evaluate the practical performance of libInterMAC, we measured the number of clock cycles used to initialise (`im_initialise()`), encrypt (`im_encrypt()`) and decrypt (`im_decrypt()`). Figure 7.8 shows the number of clock cycles per byte when encrypting files of size 1KB, 8KB, 15KB, and 50KB with libInterMAC, when either AES-GCM or ChaCha20-Poly1305 is used as the internal nonce-based AE scheme. Figure 7.9 shows the corresponding figures for decryption. These measurements were performed on a dedicated Amazon Web Services (AWS) EC2 `m4.large` instance which runs `Linux 4.14` with an `Intel Xeon E5-2676 2.4 Ghz` CPU containing the `AES-NI` instruction set and the `CLMUL` instruction set. The number of clock cycles for initialise remains constant at approximately 366k clock cycles over both internal nonce-based AE schemes and all chunk lengths, and is therefore not depicted.

We can make a number of observations from these measurements. Firstly, the number of clocks per byte is between 5 and 50 times higher when the internal nonce-based AE scheme is ChaCha20-Poly1305 compared to AES-GCM. The main reason is the availability of AES-GCM-specific CPU instructions on the machine used for benchmarking. Secondly, the closer the chunk length is to the actual message size, the more efficient IM is. This is because the message is split into fewer chunks, so that processing the message avoids multiple executions of glue code. Thirdly, there is a noticeable difference between the performance of encryption and decryption for both choices of the internal nonce-based AE scheme. The discrepancy arises because of the constant-time padding removal code used in decryption, which is forced to touch every byte of every chunk. The relative discrepancy when using ChaCha20-Poly1305 is less significant only because ChaCha20-Poly1305 is much slower than AES-GCM on the platform where the measurements were performed. The performance cost of using constant-time padding removal in libInterMAC is illustrated in Figure 7.7, which compares this option with decryption using simple non-constant-time padding removal code.

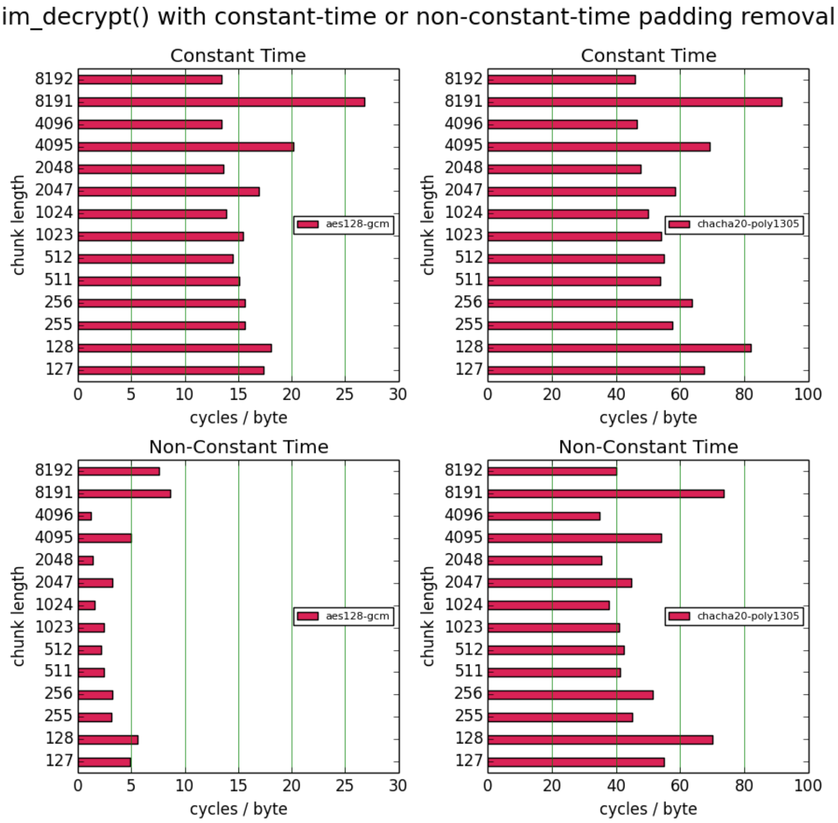

Figure 7.7: Comparing the cost of decryption using constant-time and non-constant-time padding removal in libInterMAC, with AES-GCM and ChaCha20-Poly1305 as internal nonce-based AE schemes. The constant-time option involves a significant performance penalty, especially for AES-GCM.

7.8 IM-based SSH Encryption Scheme in OpenSSH

In this section, we extend the widely-deployed SSH implementation OpenSSH with InterMAC-based SSH encryption schemes. In the following, we will denote this extension of OpenSSH as IMOpenSSH. The IM-based SSH encryption schemes are obtained using libInterMAC. Using IM-based SSH encryption schemes is as easy as using existing OpenSSH SSH encryption schemes and requires no special knowledge about IM apart from making a choice of the chunk length and the internal nonce-based AE scheme. The code for IMOpenSSH is available at [4].

We then report performance measurements on IMOpenSSH for the specific use-case of securely transferring data files between two machines using SCP (commonly an abbreviation of Secure Copy Protocol), a data transfer protocol which builds on SSH.

We have chosen SSH because Chapter 6 shows that no existing SSH encryption schemes satisfy all 4 security notions in the ciphertext fragmentation framework. SCP uses the BPP of SSH as transport protocol. It is therefore natural to use this protocol to perform performance tests. libInterMAC could be used to implement IM-based schemes in other protocols e.g. TLS. We restirct our attention to SSH to stay in-line with the theme of this thesis.

7.8.1 Implementation Details

In this section we highlight the most important implementation choices that were made during the integration of libInterMAC with OpenSSH.

Deviations from RFC 4253

Contrary to OpenSSH's native SSH-AES-GCM, SSH-Generic-EtM and SSH-ChaCha20-Poly1305 schemes discussed in Section 2.4.2, it is possible to directly apply IM to the entire encryption scope of an SSH packet. This is because the length field is not needed to locate the end of a ciphertext. However, applying IM runs into incompatibilities with the mandatory SSH padding. As described earlier, the maximum amount of padding allowed by the RFC is 255 bytes. But IM requires that the underlying plaintext be padded to be a multiple of the chunk length (plus one). Enforcing the RFC requirement would restrict the chunk length to be strictly less than 255. In addition, the SSH padding would be redundant when used with IM, as IM already adds padding up to the chunk boundary.

We chose to deviate from the SSH BPP packet format by removing the following fields: packet length field, padding length field and padding field. Removing these fields reduces the scope of encryption to just the payload. This approach was chosen for several reasons. Firstly, it reduces complexity by removing fields that no longer serve any purpose. Secondly, it removes the need to add two forms of padding. Thirdly, the encrypted packet length field has been used several times as an attack vector against the SSH protocol. Removing this field completely nullifies previous attacks and reduces the overall attack surface of the scheme. On the other hand, deviation from the SSH packet format might make it more difficult to get IM adopted in practice. However, we believe the positive aspects outweigh the negative aspects here.

Another requirement in RFC 4253 is to explicitly include the sequence number in the computation of the authentication tag. The OpenSSH SSH-AES-GCM scheme already deviated from this requirement: it instead implicitly includes the sequence number via the nonce (it is, up to an additive constant, just the low 32 bits of the AES-GCM invocation counter). IM behaves in a similar way: the IM message counter is incorporated into the nonce of the underlying nonce-based AE scheme, and is just an additive offset of the SSH sequence number as a consequence of how these are initialised and updated in an SSH connection.

Identifiers for IM-based SSH schemes

Standard SSH encryption schemes are built from an encryption scheme and a MAC algorithm that are negotiated separately during the SSH handshake protocol. The OpenSSH scheme SSH-AES-GCM, which simultaneously provides encryption and authentication, is negotiated as an encryption scheme and the MAC part of the SSH negotiation is simply ignored. All encryption schemes and MAC algorithms have named identifiers defined specifically for use in SSH. These identifiers also encode information about the key size and other parameters. The SSH negotiation process does not support negotiating additional metadata. This means that it is not possible to dynamically negotiate a chunk length for IM schemes during the SSH handshake. Instead, to fully specify an IM-based scheme in the SSH handshake context, it is necessary to define a new identifier for each different chunk length. In addition, the identifier must encode information about which internal nonce-based AE scheme is to be used in IM. In IMOpenSSH, identifiers for IM schemes are strings resulting from concatenating the following substrings (in the order appearing in the list):

- The string `"im-"`
- One string from the following set: {`"aes128-gcm-"`,`"chacha-poly-"`}
- One string from the following set:
  {`"127"`,`"128"`,`"257"`,`"256"`,`"511"`,`"512"`,`"1023"`,`"1024"`,
  `"2047"`,`"2048"`,`"4095"`,`"4096"`,`"8191"`,`"8192"`}

The first string identifies the use of an IM-based scheme. The second set of strings specifies the internal nonce-based AE scheme, while the third set specifies the chunk length of the negotiated IM scheme.

Dynamic negotiation could be introduced during key exchange by defining a new SSH key exchange method that, in addition to carrying out the shared secret establishment, also includes methods to dynamically negotiate the chunk length and internal nonce-based AE scheme. However, this would drastically increase the burden of integrating IM-based SSH encryption schemes, which is why we have chosen our initial approach of statically negotiating these parameters.

Choice of IM Parameters a and b

Recall, that a is the number of bits in the IM message counter and b is the number of bits in the IM chunk counter, and these must sum to 96; libInterMAC hard-codes $(a, b) = (64, 32)$. However, depending on context, it might be beneficial to change these parameters. For example, if $a \gg b$ then the number of messages per-key that could be securely encrypted would increase, which could be useful when encrypting many messages with few chunks. Note, changing the values of a and b might change the data limits per key, derived in Section 7.4.

7.9 Performance of IM for Secure File Transfers

SCP (Secure Copy Protocol) is a secure protocol to transfer data between two hosts. It is based on SSH and inherits the available choice of cryptographic algorithms from SSH. OpenSSH implements SCP and our integration of libInterMAC into OpenSSH allows SCP to make use of IM schemes in a seamless manner.

We used the OpenSSH implementation of SCP to carry out two sets of experiments measuring the performance and data-usage of native OpenSSH schemes and IM schemes. Specifically, we set up a client and server on two different AWS EC2 instances. For the first set of experiments, a 100MB file was transferred between two `t2.nano` AWS EC2 instances located in two different regions (EU London and US Oregon), see Figure 7.10. For the second set of experiments, a 50MB file was transferred between two `m4.large` AWS EC2 instances located in two different availability zones in the EU London region, see Figure 7.11. For both experiments, we plot the MB/s rate (computed by taking the ratio of the size of the file transferred and the median wall-time) and the median of the total volume of ciphertext. We do this for a number of IM schemes, two OpenSSH schemes based on authenticated symmetric encryption schemes (abbreviated OpenSSH AE schemes in the sequel) and an OpenSSH scheme based on CBC-mode (abbreviated OpenSSH CBC-mode scheme, in the sequel). The measurements were performed on machines running `Linux 4.14` with an `Intel Xeon E5-2676 2.4 Ghz` CPU having the `AES-NI` instruction set and the `CLMUL` instruction set.

The IM schemes suffer from substantial ciphertext expansion – we saw a 10%-30% increase compared to the raw file size. The amount of ciphertext expansion depends on how the chunk length aligns with the size of the data segments fed by the SCP application to the transport layer in SSH. The size of data segments depends on the platform and varies during file transfers, hence it is difficult to pick an optimal chunk length at the outset.

Figure 7.10 shows the results of experiments done between data centres in different regions, i.e. in a WAN setting. It indicates a relationship between the amount of ciphertext expansion and the throughput. The impact on throughput of increased ciphertext expansion on performance is low for IM schemes with a chunk length of 512 and 1024, while it tops out at around 15%-20% for IM schemes with a chunk length of 8192 (as compared to the best OpenSSH AE schemes). The OpenSSH AE schemes `aes128-gcm@` and `chacah20-poly1305@`, and the OpenSSH CBC-mode scheme `3des-cbc+hmac-md5` all have similar throughput and similar ciphertext expansion. The reason that the CBC-mode scheme achieves the same throughput as the computationally faster AE schemes is that, in the WAN setting of this experiment, they are all able to consume all the available network bandwidth and are therefore not limited by computational performance.

The results shown in Figure 7.11 were obtained on a network with a larger bandwidth capacity, effectively a LAN setting. This gives further insight into computational performance differences for each scheme and the impact on throughput. Some of the IM schemes incur significant performance hits for some chunk lengths. For example, if ChaCha20-Poly1305 is chosen as the internal nonce-based AE scheme, then the best performing IM scheme suffers a 70% performance hit when compared to OpenSSH's native SSH-ChaCha20-Poly1305 scheme. The performance difference between IM schemes using AES-GCM as the internal symmetric encryption scheme and OpenSSH's native SSH-AES-GCM scheme is less pronounced, but the IM schemes still suffer from at least a 40% decrease in performance. Furthermore, the OpenSSH SSH-AES-GCM scheme also consumes all the available bandwidth in this experiment, and so one could expect that the difference in performance would be even greater on a network with yet higher bandwidth.

IM schemes with chunk lengths $N \in \{127, 257, 511, 1023, 2047, 4095, 8191\}$ are not displayed on the charts in Figure 7.10 and Figure 7.11 because they perform similarly to the schemes with chunk length 1 greater. In view of the experiments reported for libInterMAC in Section 7.7, one might expect a difference. The reason there is not a difference is that the sizes of data segments fed by the SCP application to the transport layer in SSH are badly aligned with both choices of chunk length. Indeed, if the chunk length were chosen carefully, is it likely that an increase in performance could be obtained for the IM schemes.

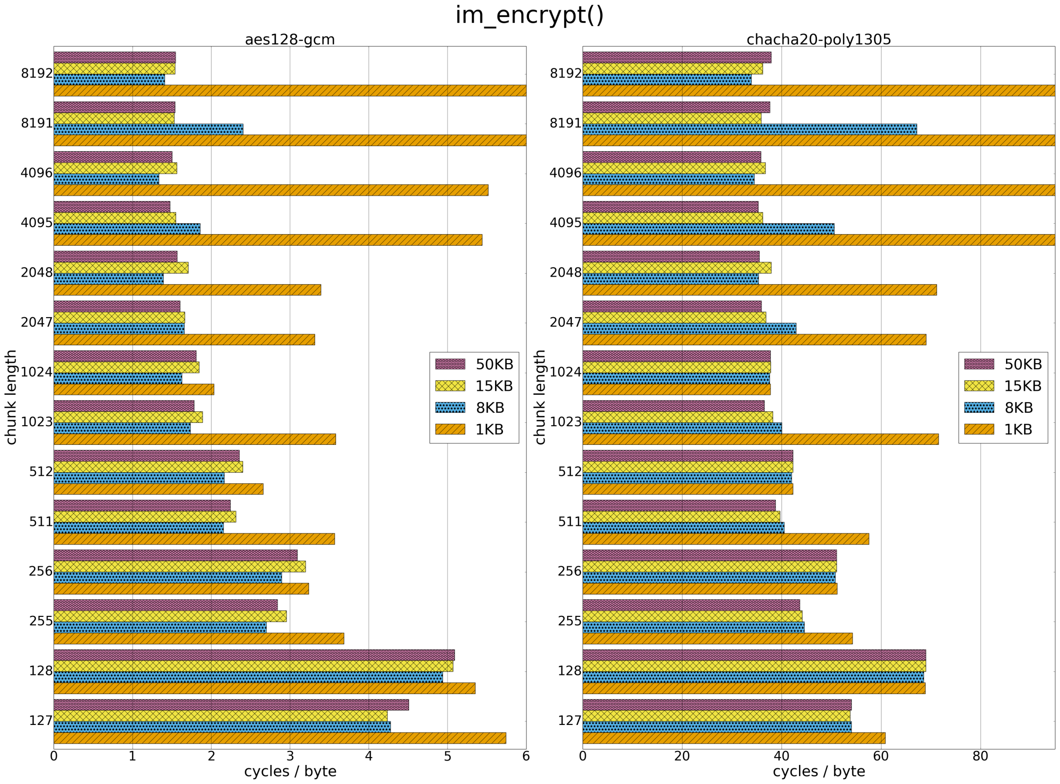

Figure 7.8: Performance measurements (lower is better) of the encrypt function (im_encrypt()) in libInterMAC for a number of chunk lengths and message lengths. Each chart shows the number of clock cycles per byte using either AES-GCM (left chart) or ChaCha20-Poly1305 (right chart) as the internal nonce-based AE scheme. The number of clock cycles per byte, for each combination of chunk length and message length (of size 1KB, 8KB, 15KB or 50KB, as indicated by four distinct label colors/patterns), is computed by taking the minimum of 25 independent averages where each average is calculated over 100 samples. AES-GCM is implemented using AES-NI and CLMUL instructions, while ChaCha20-Poly1305 is done purely in software (cf. Section 7.4). Some bars for the 1KB message category are truncated to increase readability; Left chart: the value for a chunk length of 8191/8192 is approximately 10 cycles/byte. Right chart: the value for chunk lengths of 4095/4096 or 8191/8192 are approximately 140 cycles/byte and 270 cycles/byte, respectively.

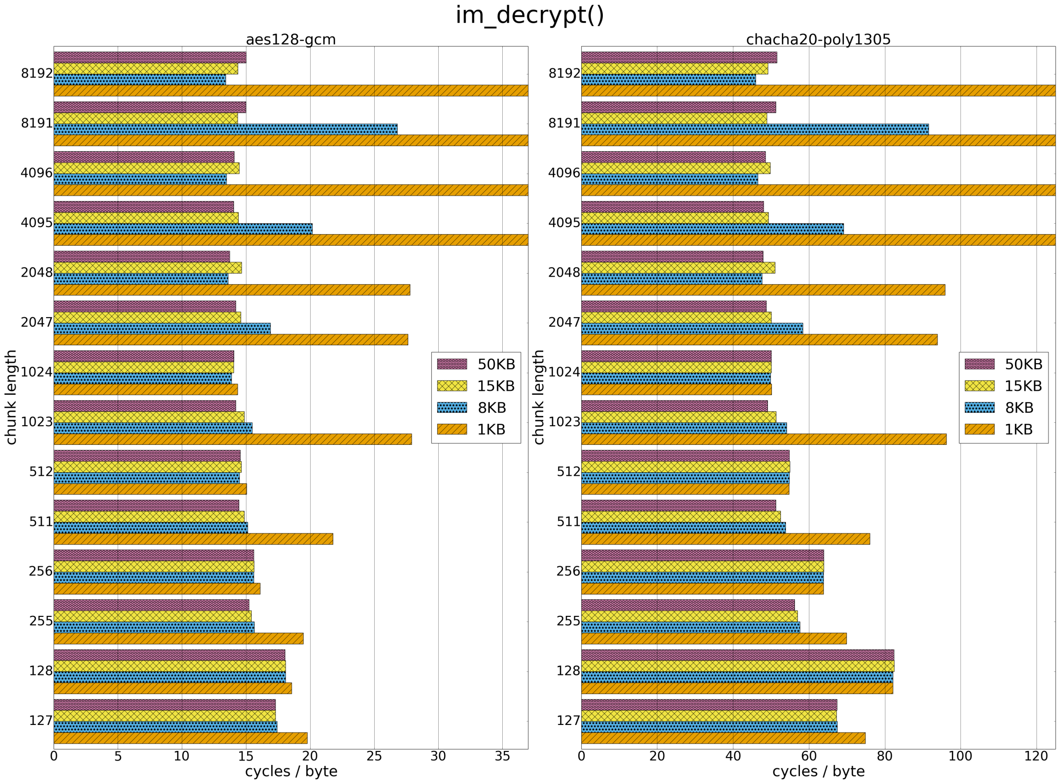

Figure 7.9: Performance measurements (lower is better) of the decrypt function (`im_decrypt()`) in libInterMAC for a number of chunk lengths and message lengths. Each chart shows the number of clock cycles per byte using either AES-GCM (left chart) or ChaCha20-Poly1305 (right chart) as the internal nonce-based AE scheme. The number of clock cycles per byte, for each combination of chunk and message length (of size 1KB, 8KB, 15KB or 50KB, as indicated by four distinct label colors/patterns), is computed by taking the minimum of 25 independent averages where each average is calculated over 100 samples. AES-GCM is implemented using AES-NI and CLMUL instructions, while ChaCha20-Poly1305 is done purely in software (cf. Section 7.4). `im_decrypt()` uses constant-time padding removal, cf. Section 7.6.1. Some bars for the 1KB message category are truncated to increase readability. Left chart: the value for chunk lengths of 4095/4096 or 8191/8192 are approximately 55 cycles/byte and 115 cycles/byte, respectively; Right chart: the value for chunk lengths of 4095/4096 or 8191/8192 are approximately 180 cycles/byte and 360 cycles/byte, respectively.

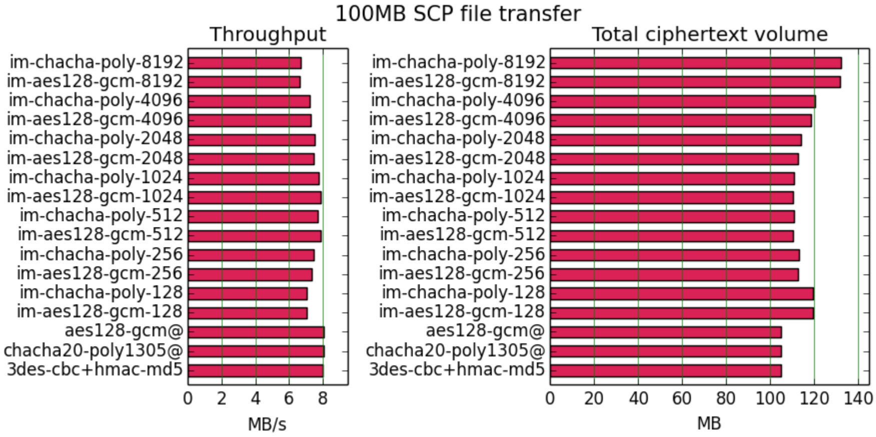

Figure 7.10: Measurements for InterMAC-based encryption schemes, OpenSSH AE-based encryption schemes and the OpenSSH AES-CBC encryption scheme using the OpenSSH version of the data copy tool SCP between two `t2.nano` AWS EC2 instances in two different regions. "im-Y-X" denotes an InterMAC-based scheme being used with Y as the internal nonce-based AE scheme, and with chunk length X bytes, @ is short-hand for @openssh. (**left chart**) Throughput in MB/s (higher is better); median over 100 samples for a 100MB file transfer for each encryption scheme. (**right chart**) Total volume of ciphertext bytes sent on the wire (lower is better); median over 100 samples for a 100MB file transfer for each SSH encryption scheme.

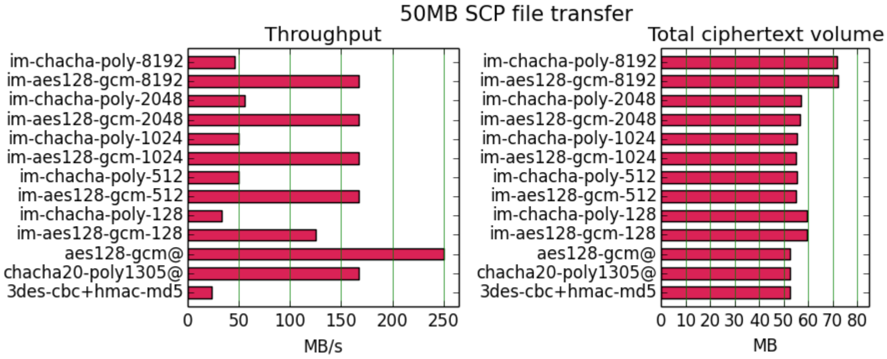

Figure 7.11: Measurements for InterMAC-based encryption schemes, OpenSSH AE-based encryption schemes and the OpenSSH AES-CBC encryption scheme using the OpenSSH version of the data copy tool SCP between two dedicated `m4.large` AWS EC2 instances in two different availability zones. "im-Y-X" denotes an InterMAC-based scheme being used with Y as the internal nonce-based AE scheme, and with chunk length X bytes, @ is short-hand for @openssh. (**left chart**) Throughput in MB/s (higher is better); median over 100 samples for a 50MB file transfer for each encryption scheme. (**right chart**) Total volume of ciphertext bytes sent on the wire (lower is better); median over 100 samples for a 50MB file transfer for each SSH encryption scheme.

Chapter 8

Concluding Remarks

In this chapter, we briefly conclude and touch on avenues for future work.

In Chapter 4, we show that there has been an increase in the diversity of SSH encryption schemes preferred by SSH servers. Specially, we saw a 27.1% increase of unique SSH encryption schemes preferred by at least one SSH server in the 2019 scan compared to the 2016 scan. We argued that such diversity is not necessarily desired. On the positive side, there has been a shift in the most preferred SSH encryption scheme. In 2016, the most preferred SSH encryption schemes were a combination of CTR-mode based encryption algorithms and HMAC based MAC algorithms. However, in 2019, the most preferred SSH encryption scheme has shifted to the native AE scheme SSH-ChaCha20-Poly1305, that provably has better security properties than any other available SSH encryption scheme, as we prove in Section 6.3). This scheme is now the default choice on about half the SSH servers we found in our 2019 scan. We attribute a part of this development to the stronger default algorithm choices implemented by OpenSSH.

Our data also show that the majority of SSH software in use is trailing behind their newest versions by at least three years. Since SSH software is often installed on devices that are hard to upgrade, this situation is predictable. On the positive side, we saw an improvement in the software age between our 2016 and 2019 scans, seeing newer software being used in the latter.

In the future, it would be interesting to gather more regular statistics on SSH servers and their use of SSH encryption schemes. Collecting more periodic data would enable a more precise study of how the SSH ecosystem evolves when newer software versions are released or when a vulnerability is discovered.

In Chapter 5, we presented three new attacks against SSH encryption schemes in OpenSSH that use CBC-mode. These new attacks affect all versions above version

5.3 of OpenSSH. We also showed that the vulnerability exploited in the Albrecht-Paterson-Watson attack also exists in Dropbear, and affects all of its versions. In the 2019 scan presented in Chapter 4, we observed that a number of SSH servers still preferring CBC-mode based SSH encryption schemes. While the number of servers preferring CBC-mode is relatively small compared to the total number of servers, there are still approximately 400,000 SSH servers that are likely to be vulnerable to one of the three new attacks or the older Albrecht-Paterson-Watson attack. In addition, the number of vulnerable servers have increased in the 2019 scan compared to the 2016 scan (factoring in the lower number of Dropbear servers found in the 2019 scan).

Section 5.8 highlights that completely mitigating the attack vector enabling these attacks, would be both complex and impact performance negatively. We, therefore, recommend that the best choice is to disable and discourage the use of CBC-mode SSH encryption schemes and, in the long run, completely remove support for CBC-mode in SSH.

In Chapter 6, we give a formal security analysis for the most important options currently implemented in OpenSSH: SSH-ChaCha20-Poly1305, SSH-Generic-EtM and SSH-AES-GCM. Our results are summarised in Table 6.1. Our proofs cover the majority of the most used SSH encryptions schemes in use by SSH servers on the Internet. In particular, we prove security properties for the most popular SSH encryption scheme SSH-ChaCha20-Poly1305. It is notable that none of the analysed schemes possess all of the security properties that one might consider desirable for SSH, namely confidentiality and integrity against an adversary with access to a fragmented decryption oracle; boundary hiding against active attacks, and; resistance to denial-of-service attacks. We attempted to improve this situation in Chapter 7.

An avenue for future work is to attempt to prove concrete security properties for SSH encryption schemes that use the original SSH method of Encrypt-and-MAC. Such proof was given earlier by Paterson and Watson [114] for CTR-mode SSH encryption scheme. However, their proof was given in their ad-hoc SSH-specific security model. It would be interesting to attempt to carry over their results to the ciphertext fragmentation model.

In Chapter 7, we introduced, analysed, implemented, and measured the performance of the modified InterMAC scheme IM. Our work brings IM, with its enhanced security properties, to the point where it could be adopted as an encryption scheme in SSH. Along the way, we have addressed many specific challenges that arise when transforming cryptographic schemes and their security properties from paper into performant code. We hope that our reference implementations libInterMAC and IMOpenSSH serve as proof that IM is viable in practice and that IM-based SSH encryption schemes in the future can be made fully available in SSH, and other applications, alongside existing options. It would be beneficial to carry out further performance testing across a wider range of applications (including interactive terminal sessions) and network conditions. In particular to investigate in more detail the effect of message sizes as chosen by the application layer and its interaction with the chunk length parameter N of IM.

We have provided an extensive discussion of potential timing side-channels in our library libInterMAC, and have attempted to remove the most egregious such side-channels via a constant-time padding removal routine. We have evaluated the performance impact of this code. We have also discussed how these side-channels relate to the BH-CPA and BH-sfCFA security notions. However, more can still be done here, including writing a strictly constant-time implementation of the entire IM decryption routine and evaluating any additional performance impact that this has. A challenging goal would be to obtain formal verification that such IM implementation is indeed constant-time, in the spirit of [8, 7]. It would also be of interest to more deeply explore how to integrate timing side-channels into the existing formal security models, as well as building formal models of what we have called reactive applications and using them to explore the limits of boundary hiding (cf. Section 3.9).

Also on the theoretical side, there has been much recent interest in obtaining security bounds for AE and AEAD schemes in the multi-user setting [25, 103, 82]. It would be interesting to see how these results can be broadened to the more complex setting of schemes supporting ciphertext fragmentation. It will also be interesting to investigate the security of IM-like schemes in the new frameworks of [125, 53] that were introduced for analysing more complex secure channel protocols like those employed in SSH.

Bibliography

[1] Farzaneh Abed, Scott R. Fluhrer, Christian Forler, Eik List, Stefan Lucks, David A. McGrew, and Jakob Wenzel. Pipelineable on-line encryption. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2014.

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Simson L. Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the usability of cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy*, pages 154–171. IEEE Computer Society Press, May 2017.

[3] Martin R. Albrecht, Torben Brandt Hansen, and Kenneth G. Paterson. libIntermac. https://github.com/himsen/libintermac, 2018. Accessed: 26/09/2019.

[4] Martin R. Albrecht, Torben Brandt Hansen, and Kenneth G. Paterson. OpenSSH extended with InterMAC-based encryption schemes. `https://github.com/himsen/intermac-openssh-portable`, 2018. Accessed: 27/09/2019.

[5] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *2009 IEEE Symposium on Security and Privacy*, pages 16–26. IEEE Computer Society Press, May 2009.

[6] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE Computer Society Press, May 2013.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt,

and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1807–1823. ACM Press, October / November 2017.

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC. In Thomas Peyrin, editor, *Fast Software Encryption – FSE 2016*, volume 9783 of *Lecture Notes in Computer Science*, pages 163–184. Springer, Heidelberg, March 2016.

[9] Elena Andreeva, Guy Barwell, Ritam Bhaumik, Mridul Nandi, Dan Page, and Martijn Stam. Turning online ciphers off. *IACR Trans. Symmetric Cryptol.*, 2017(2):105–142, 2017.

[10] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. Parallelizable and authenticated online ciphers. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology – ASIACRYPT 2013, Part I*, volume 8269 of *Lecture Notes in Computer Science*, pages 424–443. Springer, Heidelberg, December 2013.

[11] Gregory V. Bard. A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In Manu Malek, Eduardo Fernández-Medina, and Javier Hernando, editors, *SECRYPT 2006, Proceedings of the International Conference on Security and Cryptography, Setúbal, Portugal, August 7-10, 2006, SECRYPT is part of ICETE - The International Joint Conference on e-Business and Telecommunications*, pages 99–109. INSTICC Press, 2006.

[12] Gregory V. Bard. Blockwise-adaptive chosen-plaintext attack and online modes of encryption. In Steven D. Galbraith, editor, *11th IMA International Conference on Cryptography and Coding*, volume 4887 of *Lecture Notes in Computer Science*, pages 129–151. Springer, Heidelberg, December 2007.

[13] M. Bellare, T. Kohno, and C. Namprempre. The Secure Shell (SSH) Transport Layer Encryption Modes. RFC 4344 (Proposed Standard), January 2006.

[14] Mihir Bellare. Practice-oriented provable-security. In Eiji Okamoto, George I. Davida, and Masahiro Mambo, editors, *Information Security, First International Workshop, ISW '97, Tatsunokuchi, Japan, September 17-19, 1997,*

Proceedings, volume 1396 of *Lecture Notes in Computer Science*, pages 221–231. Springer, 1997.

[15] Mihir Bellare, Alexandra Boldyreva, Lars R. Knudsen, and Chanathip Namprempre. Online ciphers and the hash-CBC construction. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 292–309. Springer, Heidelberg, August 2001.

[16] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 259–274. Springer, Heidelberg, May 2000.

[17] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE Computer Society Press, October 1997.

[18] Mihir Bellare, Roch Guérin, and Phillip Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In Don Coppersmith, editor, *Advances in Cryptology – CRYPTO'95*, volume 963 of *Lecture Notes in Computer Science*, pages 15–28. Springer, Heidelberg, August 1995.

[19] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO'94*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358. Springer, Heidelberg, August 1994.

[20] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In Vijayalakshmi Atluri, editor, *ACM CCS 2002: 9th Conference on Computer and Communications Security*, pages 1–11. ACM Press, November 2002.

[21] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Trans. Inf. Syst. Secur.*, 7(2):206–241, 2004.

[22] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, Heidelberg, December 2000.

[23] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, Heidelberg, August 1994.

[24] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, Heidelberg, May / June 2006.

[25] Mihir Bellare and Björn Tackmann. The multi-user security of authenticated encryption: AES-GCM in TLS 1.3. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 247–276. Springer, Heidelberg, August 2016.

[26] Daniel J. Bernstein. The poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, Heidelberg, February 2005.

[27] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, Heidelberg, April 2006.

[28] Daniel J. Bernstein. ChaCha, a variant of Salsa20. `http://cr.yp.to/chacha/chacha-20080128.pdf`, 2008. Accessed: 27/09/2019.

[29] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: Reducing attack surface at low cost. In Carlisle Adams and Jan Camenisch, editors, *SAC 2017: 24th Annual International*

Workshop on Selected Areas in Cryptography, volume 10719 of *Lecture Notes in Computer Science*, pages 235–260. Springer, Heidelberg, August 2017.

[30] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, Heidelberg, September / October 2011.

[31] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012: 2nd International Conference on Cryptology and Information Security in Latin America*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, Heidelberg, October 2012.

[32] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459. IEEE Computer Society Press, May 2013.

[33] Ritam Bhaumik and Mridul Nandi. OleF: an inverse-free online cipher. An online SPRP with an optimal inverse-free construction. *IACR Transactions on Symmetric Cryptology*, 2016(2):30–51, 2016. `http://tosc.iacr.org/index.php/ToSC/article/view/564`.

[34] D. Bider and M. Baushke. SHA-2 Data Integrity Verification for the Secure Shell (SSH) Transport Layer Protocol. RFC 6668 (Proposed Standard), July 2012.

[35] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In Natalie Silvanovich and Patrick Traynor, editors, *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016.* USENIX Association, 2016.

[36] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. Security of symmetric encryption in the presence of ciphertext fragmentation. In David Pointcheval and Thomas Johansson, editors, *Advances in*

Cryptology – EUROCRYPT 2012, volume 7237 of *Lecture Notes in Computer Science*, pages 682–699. Springer, Heidelberg, April 2012.

[37] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. On symmetric encryption with distinguishable decryption failures. In Shiho Moriai, editor, *Fast Software Encryption – FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 367–390. Springer, Heidelberg, March 2014.

[38] Alexandra Boldyreva and Nut Taesombut. Online encryption schemes: New security notions and constructions. In Tatsuaki Okamoto, editor, *Topics in Cryptology – CT-RSA 2004*, volume 2964 of *Lecture Notes in Computer Science*, pages 1–14. Springer, Heidelberg, February 2004.

[39] Dan Boneh. The decision diffie-hellman problem. In *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, pages 48–63, 1998.

[40] Nikita Borisov, Ian Goldberg, and David A. Wagner. Intercepting mobile communications: the insecurity of 802.11. In Christopher Rose, editor, *MOBICOM 2001, Proceedings of the seventh annual international conference on Mobile computing and networking, Rome, Italy, July 16-21, 2001.*, pages 180–189. ACM, 2001.

[41] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 117–129, Tapei, Taiwan, November / December 2011. Springer, Heidelberg.

[42] Sanjit Chatterjee, Neal Koblitz, Alfred Menezes, and Palash Sarkar. Another look at tightness II: practical issues in cryptography. In Raphael C.-W. Phan and Moti Yung, editors, *Paradigms in Cryptology - Mycrypt 2016. Malicious and Exploratory Cryptology - Second International Conference, Mycrypt 2016, Kuala Lumpur, Malaysia, December 1-2, 2016, Revised Selected Papers*, volume 10311 of *Lecture Notes in Computer Science*, pages 21–55. Springer, 2016.

[43] Sanjit Chatterjee, Alfred Menezes, and Palash Sarkar. Another look at tightness. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography - 18th*

International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers, volume 7118 of *Lecture Notes in Computer Science*, pages 293–319. Springer, 2011.

[44] P. Chen, Songhwai Oh, M. Manzo, Bruno Sinopoli, C. Sharp, K. Whitehouse, O. Tolle, Jaein Jeong, P. Dutta, Jonathan Hui, Shawn Schaffert, Sukun Kim, Jay Taneja, B. Zhu, Tanya Roosta, M. Howard, David Culler, and Shankar Sastry. Instrumenting wireless sensor networks for real-time surveillance. pages 3128 – 3133, 06 2006.

[45] Community. LibTomCrypt. `https://www.libtom.net/LibTomCrypt/`, 2019. Accessed: 04/09/2019.

[46] Community. OpenSSL — cryptography and SSL/TLS toolkit. `https://www.openssl.org`, 2019. Accessed: 27/09/2019.

[47] Community. Paramiko — a python implementation of sshv2. `http://www.paramiko.org`, 2020. Accessed: 01/02/2020.

[48] Ansible Community, Ansible Inc., and Red Hat Inc. Ansible. Available at `https://www.ansible.com`, 2012.

[49] OpenBSD Community. LibreSSL. `https://www.libressl.org`, 2019. Accessed: 04/10/2019.

[50] OpenSSH community. Openssh history. `https://www.openssh.com/history.html`. Accessed: 03/10/2019.

[51] Jean-Sébastien Coron, Helena Handschuh, Marc Joye, Pascal Paillier, David Pointcheval, and Christophe Tymen. Optimal chosen-ciphertext secure encryption of arbitrary-length messages. In David Naccache and Pascal Paillier, editors, *PKC 2002: 5th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*, pages 17–33. Springer, Heidelberg, February 2002.

[52] Wei Dai. SSH2 attack. `http://www.weidai.com/ssh2-attack.txt`, 2002.

[53] Jean Paul Degabriele and Marc Fischlin. Simulatable channels: Extended security that is universally composable and easier to prove. In Thomas Peyrin

and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 519–550. Springer, Heidelberg, December 2018.

[54] Jean Paul Degabriele and Kenneth G. Paterson. Attacking the IPsec standards in encryption-only configurations. In *2007 IEEE Symposium on Security and Privacy*, pages 335–349. IEEE Computer Society Press, May 2007.

[55] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 2010: 17th Conference on Computer and Communications Security*, pages 493–504. ACM Press, October 2010.

[56] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy*, pages 463–482. IEEE Computer Society Press, May 2017.

[57] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast internet-wide scanning and its security applications. In Samuel T. King, editor, *USENIX Security 2013: 22nd USENIX Security Symposium*, pages 605–620. USENIX Association, August 2013.

[58] FailOverflow. Ps3 epic fail. `https://media.ccc.de/v/27c3-4087-en-console_hacking_2010`, Dec 2010. Chaos Computer Club. Accessed: 13/10/2019.

[59] Niels Ferguson. Authentication weaknesses in AES-GCM. `https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/cwc-gcm/ferguson2.pdf`, May 2005. Accessed: 27/09/2019.

[60] Marc Fischlin, Felix Günther, Giorgia Azzurra Marson, and Kenneth G. Paterson. Data is a stream: Security of stream-based channels. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 545–564. Springer, Heidelberg, August 2015.

[61] Ewan Fleischmann, Christian Forler, and Stefan Lucks. McOE: A family of almost foolproof on-line authenticated encryption schemes. In Anne Canteaut, editor, *Fast Software Encryption – FSE 2012*, volume 7549 of *Lecture Notes in Computer Science*, pages 196–215. Springer, Heidelberg, March 2012.

[62] Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel. Poex: A beyond-birthday-bound-secure on-line cipher. *Cryptography and Communications*, 10(1):177–193, 2018.

[63] Pierre-Alain Fouque, Antoine Joux, Gwenaëlle Martinet, and Frédéric Valette. Authenticated on-line encryption. In Mitsuru Matsui and Robert J. Zuccherato, editors, *SAC 2003: 10th Annual International Workshop on Selected Areas in Cryptography*, volume 3006 of *Lecture Notes in Computer Science*, pages 145–159. Springer, Heidelberg, August 2004.

[64] Pierre-Alain Fouque, Antoine Joux, and Guillaume Poupard. Blockwise adversarial model for on-line ciphers and symmetric encryption schemes. In Helena Handschuh and Anwar Hasan, editors, *SAC 2004: 11th Annual International Workshop on Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 212–226. Springer, Heidelberg, August 2004.

[65] Pierre-Alain Fouque, Gwenaëlle Martinet, and Guillaume Poupard. Practical symmetric on-line encryption. In Thomas Johansson, editor, *Fast Software Encryption – FSE 2003*, volume 2887 of *Lecture Notes in Computer Science*, pages 362–375. Springer, Heidelberg, February 2003.

[66] Markus Friedl, Jason McIntyre, Damien Miller, Niels Provos, Theo de Raadt, Tim Rice, Kevin Steves, and Darren Tucker. Openssh. https://www.openssh.com.

[67] Markus Friedl, Jason McIntyre, Damien Miller, Niels Provos, Theo de Raadt, Tim Rice, Kevin Steves, and Darren Tucker. Openssh version 4.7 release notes. http://www.openssh.com/txt/release-4.7, 2007.

[68] Markus Friedl, Jason McIntyre, Damien Miller, Niels Provos, Theo de Raadt, Tim Rice, Kevin Steves, and Darren Tucker. Openssh version 6.2 release notes. http://www.openssh.com/txt/release-6.2, 2015.

[69] Markus Friedl, Jason McIntyre, Damien Miller, Niels Provos, Theo de Raadt, Tim Rice, Kevin Steves, and Darren Tucker. Openssh version 6.5 release notes. `http://www.openssh.com/txt/release-6.5`, 2015.

[70] Markus Friedl, Jason McIntyre, Damien Miller, Niels Provos, Theo de Raadt, Tim Rice, Kevin Steves, and Darren Tucker. Openssh version 7.3 release notes. `https://www.openssh.com/txt/release-7.3`, 2016.

[71] Markus Friedl, Jason McIntyre, Damien Miller, Niels Provos, Theo de Raadt, Tim Rice, Kevin Steves, and Darren Tucker. Openssh version 7.6 release notes. `https://www.openssh.com/txt/release-7.6`, 2017.

[72] Markus Friedl, Jason McIntyre, Damien Miller, Niels Provos, Theo de Raadt, Tim Rice, Kevin Steves, and Darren Tucker. Openssh version 7.7 release notes. `https://www.openssh.com/txt/release-7.7`, 2018.

[73] Markus Friedl, Jason McIntyre, Damien Miller, Niels Provos, Theo de Raadt, Tim Rice, Kevin Steves, and Darren Tucker. Openssh version 8.0 release notes. `https://www.openssh.com/txt/release-8.0`, 2019.

[74] Markus Friedl, Jason McIntyre, Damien Miller, Niels Provos, Theo de Raadt, Tim Rice, Kevin Steves, and Darren Tucker. Openssh version 8.1 release notes. `https://www.openssh.com/txt/release-8.1`, 2019.

[75] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th Annual Symposium on Foundations of Computer Science*, pages 464–479. IEEE Computer Society Press, October 1984.

[76] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the cryptographic applications of random functions. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 276–288. Springer, Heidelberg, August 1984.

[77] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[78] Shafi Goldwasser and Silvio Micali. Probabilistic encryption &amp; how to play mental poker keeping secret all partial information. In *Proceedings of*

the Fourteenth Annual ACM Symposium on Theory of Computing, STOC '82, pages 365–377, New York, NY, USA, 1982. ACM.

[79] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *14th Annual ACM Symposium on Theory of Computing*, pages 365–377. ACM Press, May 1982.

[80] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[81] Viet Tung Hoang, Reza Reyhanitabar, Phillip Rogaway, and Damian Vizár. Online authenticated-encryption and its nonce-reuse misuse-resistance. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 493–517. Springer, Heidelberg, August 2015.

[82] Viet Tung Hoang, Stefano Tessaro, and Aishwarya Thiruvengadam. The multi-user security of GCM, revisited: Tight bounds for nonce randomization. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1429–1440. ACM Press, October 2018.

[83] R. Housley. Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms. RFC 7696 (Best Current Practice), November 2015.

[84] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018.

[85] K. Igoe and J. Solinas. AES Galois Counter Mode for the Secure Shell Transport Layer Protocol. RFC 5647 (Informational), August 2009.

[86] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-25, Internet Engineering Task Force, January 2020. Work in Progress.

[87] Matt Johnson. Dropbear. `https://matt.ucc.asn.au/dropbear/dropbear.html`.

[88] Matt Johnson. Dropbear change log. https://matt.ucc.asn.au/dropbear/CHANGES. Accessed: 03/10/2019.

[89] Antoine Joux. Authentication failure in NIST version of GCM. https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/joux_comments.pdf, 2006. Accessed: 26/09/2019.

[90] Antoine Joux, Gwenaëlle Martinet, and Frédéric Valette. Blockwise-adaptive attackers: Revisiting the (in)security of some provably secure encryption models: CBC, GEM, IACBC. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 17–30. Springer, Heidelberg, August 2002.

[91] Charanjit S. Jutla. Encryption modes with almost free message integrity. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 529–544. Springer, Heidelberg, May 2001.

[92] B. Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315 (Informational), March 1998.

[93] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* Chapman & Hall/CRC, 2nd edition, 2014.

[94] Christopher A. Kent and Jeffrey C. Mogul. Fragmentation considered harmful. *SIGCOMM Comput. Commun. Rev.*, 25(1):75–87, January 1995.

[95] Neal Koblitz and Alfred Menezes. Another look at ”provable security”. II. In Rana Barua and Tanja Lange, editors, *Progress in Cryptology - INDOCRYPT 2006, 7th International Conference on Cryptology in India, Kolkata, India, December 11-13, 2006, Proceedings*, volume 4329 of *Lecture Notes in Computer Science*, pages 148–175. Springer, 2006.

[96] Tadayoshi Kohno. Attacking and repairing the winZip encryption scheme. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick McDaniel, editors, *ACM CCS 2004: 11th Conference on Computer and Communications Security*, pages 72–81. ACM Press, October 2004.

[97] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of age: A longitudinal study of TLS deployment. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31 - November 02, 2018*, pages 415–428. ACM, 2018.

[98] T. Krovetz. UMAC: Message Authentication Code using Universal Hashing. RFC 4418 (Informational), March 2006.

[99] Adam Langley and Wan-Teh Chang. ChaCha20 and Poly1305 based Cipher Suites for TLS. Internet-Draft draft-agl-tls-chacha20poly1305-04, Internet Engineering Task Force, November 2013. Work in Progress.

[100] Nate Lawson. Keyczar library. `https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/`, 2009. Accessed: 02/10/2019.

[101] S. Lehtinen and C. Lonvick. The Secure Shell (SSH) Protocol Assigned Numbers. RFC 4250 (Proposed Standard), January 2006.

[102] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer, Heidelberg, August 2002.

[103] Atul Luykx, Bart Mennink, and Kenneth G. Paterson. Analyzing multi-key security degradation. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 575–605. Springer, Heidelberg, December 2017.

[104] Atul Luykx and Kenneth G. Paterson. Limits on authenticated encryption use in TLS. `http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf`, 2017. Accessed: 27/09/2019.

[105] Gordon Fyodor Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning.* Insecure, 2009.

[106] Developer mailing list. Openssh. `https://lists.mindrot.org/pipermail/openssh-unix-dev/2001-February/004650.html`.

[107] David McGrew and John Viega. The galois/counter mode of operation (gcm). Submission to NIST Modes of Operation Process, 2004.

[108] Alfred Menezes. Another look at provable security (invited talk). In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, page 8. Springer, Heidelberg, April 2012.

[109] D. Miller and P. Valchev. The use of umac in the ssh transport layer protocol. `https://tools.ietf.org/html/draft-miller-secsh-umac-01`, September 2007.

[110] Damien Miller. ChaCha20 and Poly1305 in OpenSSH. `http://blog.djm.net.au/2013/11/chacha20-and-poly1305-in-openssh.html`, Nov 2013. Djm's personal webblog. Accessed: 13/10/2019.

[111] Damien Miller and Simon Josefsson. The chacha20-poly1305@openssh.com authenticated encryption cipher. Internet-Draft draft-josefsson-ssh-chacha20-poly1305-openssh-00, Internet Engineering Task Force, November 2015. Work in Progress.

[112] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton. Reconsidering generic composition. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 257–274. Springer, Heidelberg, May 2014.

[113] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539 (Informational), May 2015.

[114] Kenneth G. Paterson and Gaven J. Watson. Plaintext-dependent decryption: A formal security treatment of SSH-CTR. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 345–361. Springer, Heidelberg, May / June 2010.

[115] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864.

[116] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[117] Gordon Procter. A security analysis of the composition of ChaCha20 and Poly1305. *IACR Cryptology ePrint Archive*, 2014:613, 2014.

[118] Arnezami (pseudonym), Robinsod (pseudonym), and XboxHacker community. Xbox 360 timing attack. `https://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack`, 2007. Accessed: 02/10/2019.

[119] E. Rescorla. The transport layer security (TLS) protocol version 1.3. RFC 8446, August 2018.

[120] Phil Rogaway. Problems with proposed ip cryptography. `http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt`, April 1995.

[121] Phillip Rogaway. Practice-oriented provable security and the social construction of cryptography. *Unpublished essay* (2009).

[122] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002: 9th Conference on Computer and Communications Security*, pages 98–107. ACM Press, November 2002.

[123] Phillip Rogaway. Nonce-based symmetric encryption. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 348–359. Springer, Heidelberg, February 2004.

[124] Phillip Rogaway and Haibin Zhang. Online ciphers from tweakable blockciphers. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 237–249. Springer, Heidelberg, February 2011.

[125] Phillip Rogaway and Yusi Zhang. Simplifying game-based definitions - indistinguishability up to correctness and its application to stateful AE. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 3–32. Springer, Heidelberg, August 2018.

[126] Todd Sabin. Vulnerability in Windows NT's SYSKEY encryption. `https://marc.info/?l=ntbugtraq&m=94537191024690`, 1999. Accessed: 13/10/2019.

[127] David Tomaschik. CVE-2019-10071. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10071`, 2019. Accessed: 02/10/2019.

[128] Linus Torvalds, Junio Hamano, et al. Git. Available at `https://git-scm.com/`, 2005.

[129] Patrick P. Tsang, Rouslan V. Solomakhin, and Sean W. Smith. Authenticated Streamwise On-line Encryption. Technical Report TR2009-640, Dartmouth College, Computer Science, Hanover, NH, March 2009.

[130] Andrew M. White, Austin R. Matthews, Kevin Z. Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted VoIP conversations: Hookt on fon-iks. In *2011 IEEE Symposium on Security and Privacy*, pages 3–18. IEEE Computer Society Press, May 2011.

[131] Hongjun Wu. The misuse of RC4 in microsoft word and excel. *IACR Cryptology ePrint Archive*, 2005:7, 2005.

[132] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard), January 2006.

[133] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254 (Proposed Standard), January 2006.

[134] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[135] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), January 2006. Updated by RFC 6668.

[136] Liyang Yu, Neng Wang, and Xiaoqiao Meng. Real-time forest fire detection with wireless sensor networks. volume 2, pages 1214 – 1217, 10 2005.