IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2019, No. 1, pp. 46–83. DOI:10.13154/tosc.v2019.i1.46-83

libInterMAC: Beyond Confidentiality and Integrity in Practice

Martin R. Albrecht, Torben Brandt Hansen and Kenneth G. Paterson

Royal Holloway, University of London, London, United Kingdom
{martin.albrecht,torben.hansen.2015,kenny.paterson}@rhul.ac.uk

Abstract. Boldyreva et al. (Eurocrypt 2012) defined a fine-grained security model capturing ciphertext fragmentation attacks against symmetric encryption schemes. The model was extended by Albrecht et al. (CCS 2016) to include an integrity notion. The extended security model encompasses important security goals of SSH that go beyond confidentiality and integrity to include length hiding and denial-of-service resistance properties. Boldyreva et al. also defined and analysed the InterMAC scheme, while Albrecht et al. showed that InterMAC satisfies stronger security notions than all currently available SSH encryption schemes. In this work, we take the InterMAC scheme and make it fully ready for use in practice. This involves several steps. First, we modify the InterMAC scheme to support encryption of arbitrary length plaintexts and we replace the use of Encrypt-then-MAC in InterMAC with modern nonce-based authenticated encryption. Second, we describe a reference implementation of the modified InterMAC scheme in the form of the library libInterMAC. We give a performance analysis of libInterMAC. Third, to test the practical performance of libInterMAC, we implement several InterMAC-based encryption schemes in OpenSSH and carry out a performance analysis for the use-case of file transfer using SCP. We measure the data throughput and the data overhead of using InterMAC-based schemes compared to existing schemes in OpenSSH. Our analysis shows that, for some network set-ups, using InterMAC-based schemes in OpenSSH only moderately affects performance whilst providing stronger security guarantees compared to existing schemes.

Keywords: fragmentation attack · SSH · Authenticated Encryption · crypto library · implementation · performance analysis

1 Introduction

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 IND-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, specifically the Binary Packet Protocol (BPP), the component of the SSH protocol suite specifying data transfer. In 2002 Bellare et al. [BKN02] 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. [APW09] presented a plaintext-recovery attack against CBC mode with random IVs in SSH, a case covered by the proof. In 2016 further attacks were found against a patched version of the CBC mode construction used in OpenSSH [ADHP16]. The attacks of [APW09, ADHP16] exploit 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.

Received: 2018-11-23, Accepted: 2019-01-23, Published: 2019-03-08

They also exploit 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. [BKN02], do not cater for this kind of *ciphertext fragmentation attack*. Fragmentation of ciphertexts has also been used to mount attacks against IPSec [DP10].

The setting of symmetric encryption schemes supporting ciphertext fragmentation was formulated and thoroughly analysed by Boldyreva et al. [BDPS12], with the aim of formalising exactly what the security goals for the SSH BPP should be. They introduced confidentiality notions for this setting, IND-sfCFA, as well as two more advanced notions capturing the idea that an adversary should not be able to tell where the boundaries between distinct packets lie ("boundary hiding", BH-CPA and BH-sfCFA for the passive and active settings, respectively) and the idea that an adversary should not be able to make a receiver "hang" in the middle of a decryption operation, consuming large amounts of data without outputting anything ("Denial-of-Service" security, DOS-sfCFA). The boundary hiding notions are suggested by SSH's use of encrypted length fields and lack of per-packet metadata, indicating that the original designers of SSH wished a long sequence of SSH packets to look just like a string of random bits. The DOS-sfCFA notion is motivated by SSH's interactive session mode, SSH's support for relatively large packet lengths, and the manipulability of the length field by an active attacker. An integrity notion, IND-sfCTF, for symmetric encryption schemes supporting ciphertext fragmentation was later defined in [ADHP16].

It is notable that none of the symmetric encryption schemes currently supported in SSH (nor in the leading OpenSSH implementation of SSH) achieve the four strongest properties in combination (IND-sfCFA, IND-sfCTF, BH-sfCFA, DOS-sfCFA), see [ADHP16]. For example, the now-default scheme in OpenSSH 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 [Mil13]. The OpenSSH 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. A complete summary of the encryption schemes available in OpenSSH and their security properties can be found in [ADHP16, Table 1].

The above discussion begs the question: can one do better, achieving all four of the strong security notions formalised by Boldyreva et al. [BDPS12], and at what cost? In fact, [BDPS12] already defined a scheme, InterMAC, that possesses the four security notions: IND-sfCFA, IND-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). It is claimed by Boldyreva et al. [BDPS12] that InterMAC is efficient in practice. We interpret this statement to mean that InterMAC demonstrates that all four security properties can be met in practice with *low overhead* and so would enhance security with no significant decrease in performance compared to existing schemes.

1.1 Our Contributions

Reflecting on the discussion above, it is evident that there is a need to implement and deploy schemes that satisfy the advanced security notions desired (but not currently met) by the designers of the SSH protocol. To this end, we started from the InterMAC scheme and investigated the claim that InterMAC only requires low overhead. Specifically, we implemented a C library libInterMAC [AHP18a], providing two options for choosing the internal encryption scheme to replace InterMAC's Encrypt-then-MAC approach, namely AES-GCM and ChaCha20-Poly1305. The library can be easily extended with other options. We tested the performance of the library, see Figures 1 and Figure 2, and used it to develop new InterMAC-based encryption schemes for OpenSSH. Section 4.7 gives a detailed discussion of the performance of libInterMAC. We compare these new schemes with existing OpenSSH options to assess their impact on data overhead and latency. We also provide a detailed discussion of the challenges of implementing libInterMAC in a constant time manner (in an effort to reduce the side-channel leakages) and the impact of this on performance. These considerations go beyond the formal modelling of security properties considered in prior work [BDPS12, ADHP16].

Most notably, the process of implementing InterMAC and InterMAC-based encryption schemes for SSH and addressing concerns arising in practice required us to modify the InterMAC scheme and the SSH packet format in significant ways:

- The original InterMAC scheme only supported plaintexts whose *bit*-length is a multiple of some chunk length. For reasons of practicality, we focus on *byte*-oriented plaintexts. But to support the arbitrary length inputs encountered in practice, we add alternating-byte padding to the scheme. Unfortunately, adding this feature significantly increases the ciphertext expansion, since padding up to a multiple of N bytes is required in order to preserve BH-sfCFA security whilst achieving DOS-sfCFA security. This padding has a negative impact on the performance, as we describe below.

- For efficiency and convenience, we decided to employ native nonce-based AE schemes to process chunks, instead of using the original Encrypt-then-MAC construction in InterMAC with its two separate processing steps. This required a number of design decisions to be made concerning signaling of the last chunk, and how to select nonces for the underlying AE scheme so as to incorporate both a message counter and a chunk counter to prevent reordering and deletion attacks.

- When InterMAC is used in the SSH context, a number of the original fields in the SSH packet format are made redundant. We therefore made the decision to modify the SSH packet format to eliminate the now unneeded fields, saving some overhead in the encoding of SSH data. The packet processing of our InterMAC-based scheme is implemented in a standalone code path in OpenSSH, bypassing the complex packet processing code (see [ADHP16] for examples of security issues arising in that part of the code). Any application using OpenSSH with one of our new InterMAC-based schemes would be oblivious to this change.

Our performance analysis shows that our InterMAC schemes do suffer from a non-negligible ciphertext expansion, up to 30% for the measured InterMAC schemes, compared to native schemes, see Figures 3 and 4. This is as a result of the desire to achieve BH-sfCFA security and DOS-sfCFA security simultaneously. However, the behaviour is not uniform over all InterMAC schemes and depends on the type of data being sent (particularly, the length of messages supplied by the application running over SSH), network configuration and the chunk length parameter N. In particular, if the data chunks being processed by SSH align badly with the chunk length, then a large amount of padding is needed to

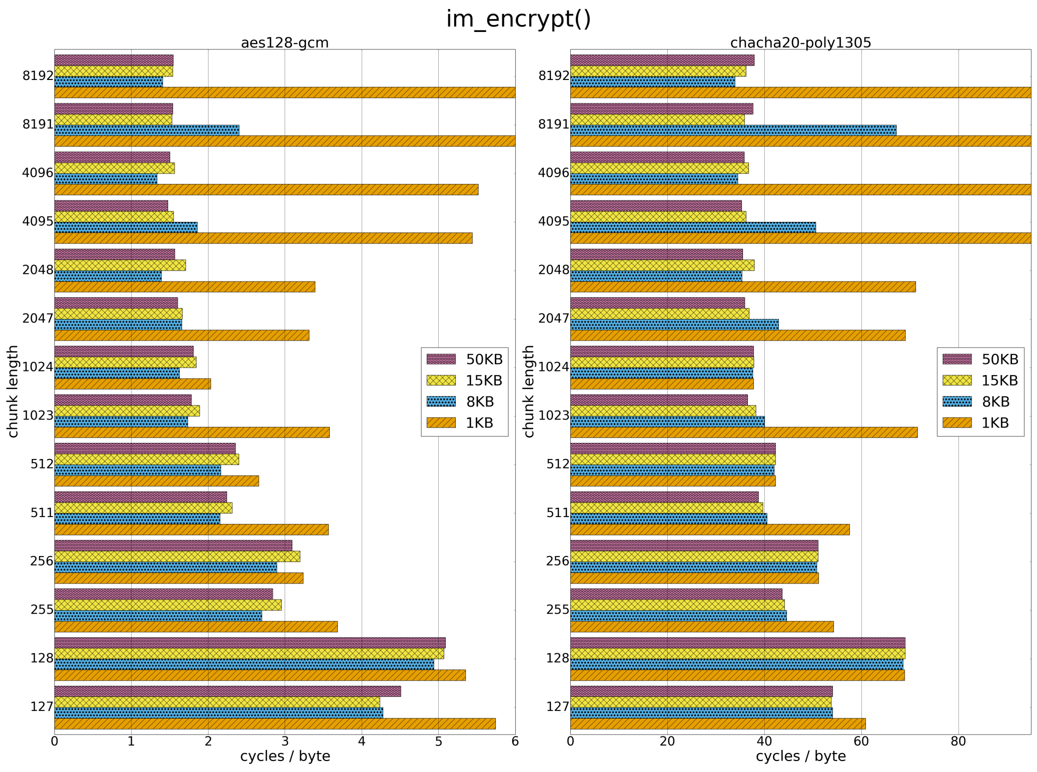

Figure 1: Performance measurements (lower is better) of the encrypt function (`im_encrypt()`) in libInterMAC for a number of chunk lengths and message lengths. Each chart shows the number of clock cycles per byte using either AES-GCM (left chart) or ChaCha20-Poly1305 (right chart) as the internal nonce-based 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 4.4). 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.

push the message size up to a multiple of the chunk length. This brings up the important question of to what extent (and how) cryptographic algorithms and their security properties should be designed and then tuned to cater for the many diverse environments that may exist in practice.

On the other hand, when performance is measured purely based on throughput for large file transfers, then, depending on the network setting, the performance of the InterMAC schemes may not significantly deviate from the existing schemes in OpenSSH. For example, when using SCP to transfer a 100MB file in a WAN setting, simulated by transferring the file between two `t2.nano` AWS EC2 instances in two different regions, one cannot distinguish the throughput of InterMAC schemes from native OpenSSH schemes, see Figure 3. However, when transferring a 50MB file in a LAN setting, simulated by transferring the file between two `m4.large` AWS EC2 instances between two different availability zones (in the same region), there is a significant difference in throughput between InterMAC schemes and

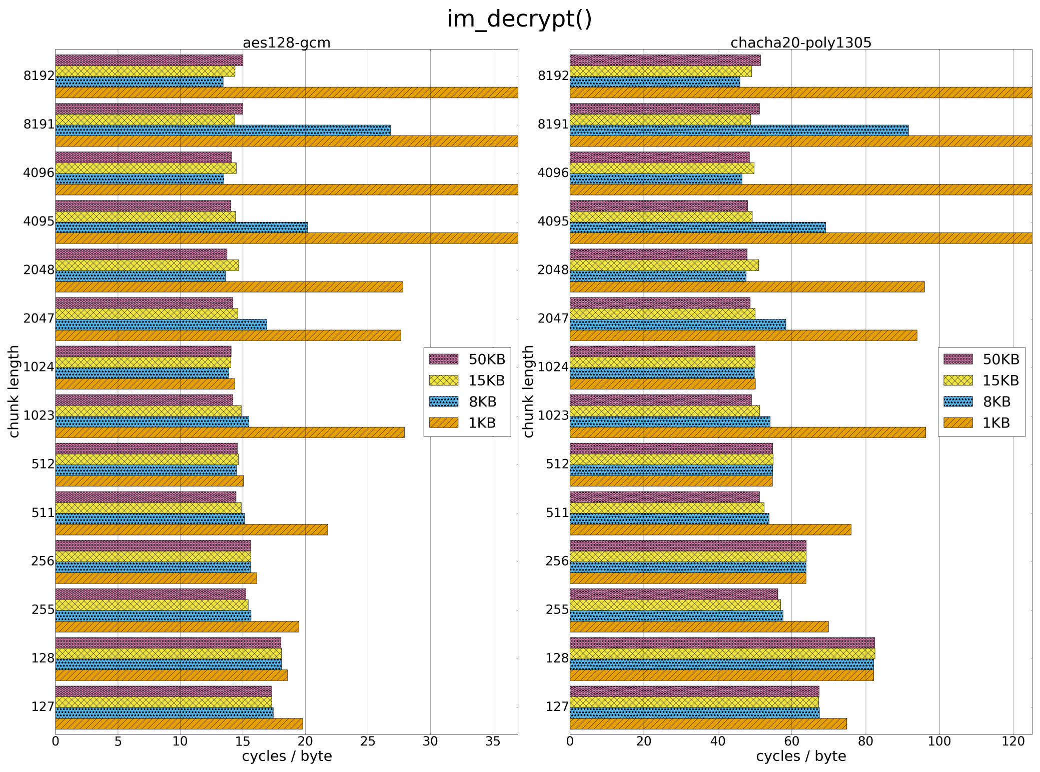

Figure 2: Performance measurements (lower is better) of the decrypt function (`im_decrypt()`) in libInterMAC for a number of chunk lengths and message lengths. Each chart shows the number of clock cycles per byte using either AES-GCM (left chart) or ChaCha20-Poly1305 (right chart) as the internal nonce-based 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 4.4). `im_decrypt()` uses constant-time padding removal, cf. Section 4.6.1. 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.

the native OpenSSH schemes, see Figure 4. In the former case, the overall performance is bandwidth-limited rather than computation-limited. The reverse is true in the latter case: we were not able to saturate the available bandwidth, and computation is clearly the bottleneck, with the difference in throughput for the different schemes becoming clearly visible. These two experiments indicate, that depending on the application and network setting, greater security can be achieved than that provided by the existing OpenSSH schemes whilst suffering only a small decrease in performance. For more details of our performance study, see Section 5.3.

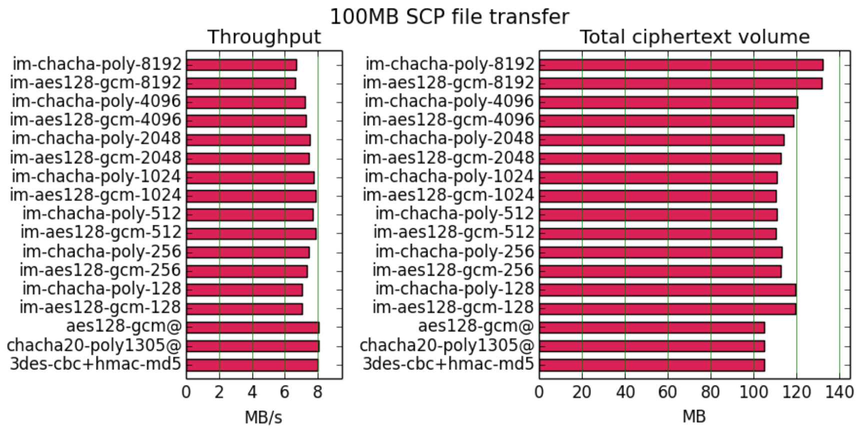

Figure 3: Measurements for InterMAC-based encryption schemes, OpenSSH AEAD-based encryption schemes and the OpenSSH AES-CBC encryption scheme using the OpenSSH version of the data copy tool SCP between two `t2.nano` AWS EC2 instances in two different regions. "im-Y-X" denotes InterMAC-based scheme being 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 encryption scheme.

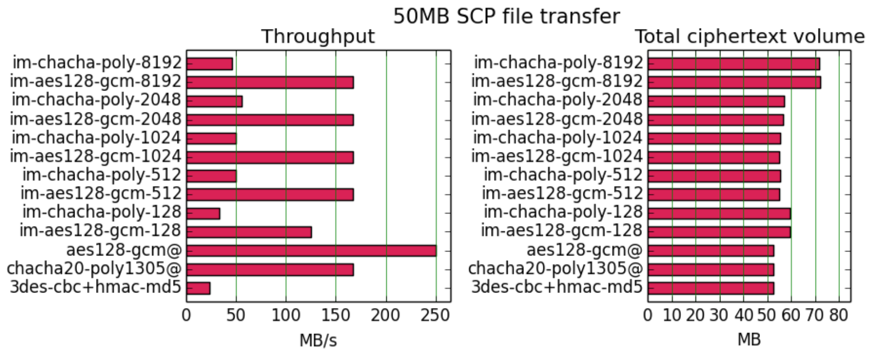

Figure 4: Measurements for InterMAC-based encryption schemes, OpenSSH AEAD-based encryption schemes and the OpenSSH AES-CBC encryption scheme using the OpenSSH version of the data copy tool SCP between two dedicated `m4.large` AWS EC2 instances in two different availability zones. "im-Y-X" denotes InterMAC-based 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 encryption scheme.

1.2 Paper Organisation

In Section 2, we cover preliminaries, including formal definitions of the security notions for the fragmented setting that we target with InterMAC. Section 3 describes in pseudo-code the original InterMAC scheme and our modified version of it for implementation. In Section 4, we discuss details of our implementation (libInterMAC) of InterMAC. Section 5 discusses integration of libInterMAC with OpenSSH and our experiments with file transfers using SCP running over libInterMAC. Section 6 presents our conclusions.

1.3 Further Related Work

The concept of symmetric encryption supporting ciphertext fragmentation as introduced in [BDPS12] and used here is related to – but distinct from – online Authenticated Encryption (oAE) [FFL12, HRRV15]. One way to understand the difference is that oAE seeks to extend the definition of AE to study its security in the situation where the AE encryption (and possibly decryption) operation is online and operates on chunks of data of some fixed size, while symmetric encryption supporting ciphertext fragmentation rather takes as its starting point a set of properties desired of a secure channel of a particular type, possibly realising constructions using AE (as we do here).

A second related concept is that of unverified release of plaintext [ABL+14], which is also an issue arising in the specific context of online AE, but which relates to per-chunk processing of ciphertexts. The relations between different notions of security in this sphere were investigated in [BPS15] and [EV16].

Finally, [FGMP15] considers a similar, yet distinct concept to symmetric encryption supporting ciphertext fragmentation, namely stream-based channels. This is tailored to protocols like TLS in which a TCP-like streaming interface is presented to both encryption and decryption, whereas, in our setting, only the decryption interface behaves in this way.

2 Preliminaries

2.1 Bits vs. Bytes

In this work, we have chosen to take a byte-oriented approach when describing algorithms. That is, operations performed in an algorithm are performed byte-wise. However, because we want to be compatible with previous works we keep standard bit-oriented security definitions.

Our main reason for choosing byte-oriented notation is the strong practical view we take in this work. Odd bit lengths are a significant headache for implementers and are seldom used in network protocols, SSH being a prime example of a byte-oriented protocol. As a result, describing data strings, etc., in bits imposes extra work and creates confusion, but yields no practical gain.

2.2 Notation

For algorithms $A_1, A_2, \dots$ we let $\mathcal{A}^{A_1, A_2, \cdots}$ denote the output after executing $\mathcal{A}$ with oracle access to $A_1, A_2, \dots$ and 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 $s \leftarrow_R S$ 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. ε denotes the empty string. For two strings v, w, consisting of either bits or bytes, we use the following notation:

– $v \parallel w$ denotes the concatenation of strings v and w.
– $|v| / |v|_\mathsf{B}$ denotes the size of string v counted in bits/bytes.
– $v \mid w$ denotes the bit-wise OR operation between v and w.
– $v \oplus w$ denotes the bit-wise XOR operation between v and w.
– $v \preceq w$ denotes the prefix predicate and returns true if and only if there exists $u \in \{0,1\}^*$ such that $w = v \parallel u$.

- $v \,\%\, w$ denotes the (unique) substring 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 bit or byte string such that $p \preceq v$ and $p \preceq w$).
- $v[i]/v[i]_{\mathsf{B}}$ denotes the ith bit/byte of v.
- $v[i:j]/v[i:j]_{\mathsf{B}}$ denotes the substring from bit/byte i to bit/byte j (inclusive) of v.

For a list $\mathcal{L}$, we use $\mathcal{L} = []$ to denote the initialisation of $\mathcal{L}$ to the empty list and let $\mathcal{L}.\mathsf{append}(L)$ denote appending L to the list $\mathcal{L}$. The check $L \in \mathcal{L}$ returns true if L is in the list $\mathcal{L}$ and false otherwise. $\mathcal{L}[p \dots q]$ denotes the sublist $[\mathcal{L}[p], \mathcal{L}[p+1], \dots, \mathcal{L}[q]]$. For a vector $\mathbf{m} = (m_1, m_2, \dots, m_r)$ or list $\mathcal{L} = [m_1, m_2, \dots, m_r]$, we let $\|(\mathcal{L})$ and $\|(\mathbf{m})$ denote the concatenation $m_1 \parallel m_2 \parallel \cdots \parallel m_r$.

Recall, that the definitions below are all bit-oriented, while the algorithms descriptions, which will appear later, are byte-oriented.

2.3 Nonce-Based Symmetric Encryption Scheme

Our definition of a nonce-based authenticated encryption (nAE) scheme is inspired by Namprempre et al. [NRS14]. We will use such schemes in a generic way in the sequel when proposing our implementation-oriented version of InterMAC. In our security definitions, the adversary success is quantified in terms of the adversary's resources, denoted by R. The resources of interest are usually the number of queries of different types, the size of the queries and the running time of the adversary. See [BGR95, BKR94, BR94] for details of this approach.

Definition 1 (nAE scheme)**.** A nonce-based symmetric encryption scheme is a triple $\mathsf{nSE} = (\mathsf{K}, \mathsf{Enc}, \mathsf{Dec})$, with associated message space $\mathcal{M} \subseteq \{0,1\}^*$, ciphertext space $\mathcal{C} \subseteq \{0,1\}^*$, and nonce space $\mathcal{N} \subseteq \{0,1\}^*$. For $\mathsf{k} \leftarrow \mathsf{K}$, we write $C \leftarrow \mathsf{Enc}_{\mathsf{k}}(M, N)$ where $(M, N, C) \in \mathcal{M} \times \mathcal{N} \times \mathcal{C}$ and $M \leftarrow \mathsf{Dec}_{\mathsf{k}}(C, N)$ where $(C, N, M) \in \mathcal{C} \times \mathcal{N} \times (\mathcal{M} \cup \{\perp\})$ (with $\perp \notin \mathcal{M}$). See Namprempre et al. [NRS14] for details on further requirements on Enc and Dec.

Note that Namprempre et al. actually define nAE schemes with additional data. We omit additional data in this paper (because we do not make any use of it). The security definition we use for an nAE scheme is also inspired by Namprempre et al. [NRS14].

Definition 2 (nAE advantage)**.** Let $\mathsf{nSE} = (\mathsf{K}, \mathsf{Enc}, \mathsf{Dec})$, with associated message space $\mathcal{M} \subseteq \{0,1\}^*$, ciphertext space $\mathcal{C} \subseteq \{0,1\}^*$, and nonce space $\mathcal{N} \subseteq \{0,1\}^*$ be an nAE scheme. Let ENC, DEC, \$ and $\perp$, all being initialised by INI, be the algorithms defined in Fig. 5. For any adversary $\mathcal{A}$ we define its nAE advantage as:

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

An adversary $\mathcal{A}$ is said to be (η, R)-nAE secure if for any adversary $\mathcal{A}$ using resources at most R, its nAE advantage $\mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A})$ is bounded by η.

Informally, the definition measures an adversary's ability to distinguish an encryption/decryption oracle pair from a pair of oracles that return random bits and $\perp$ (error). We assume that all the nonce-based symmetric encryption schemes used in this paper have only one possible error message. This simplifies our treatment and is well-reflected in practice for the AE schemes we use internally in our constructions (but note that our syntax for symmetric encryption schemes supporting ciphertext fragmentation in the next section do not have this restriction).

```
alg. $\mathsf{INI}$
$\mathsf{k} \leftarrow_R \mathsf{K}$
$\mathcal{L}_E = []$
$\mathcal{L}_D = []$
return
```

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

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

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

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

Figure 5: Algorithms for defining nAE advantage.

2.4 Symmetric Encryption in the Presence of Ciphertext Fragmentation

Standard security models for symmetric encryption schemes lack the granularity to correctly capture the use of symmetric encryption schemes in real world protocols. TLS, SSH and IPsec, which all heavily rely on the security of symmetric encryption schemes, are often implemented on top of TCP or other lower-level protocols that fragment data in ways that are uncontrollable and unpredictable. As a consequence, ciphertexts might arrive at a receiver in an arbitrarily fragmented fashion and before decryption can happen, these fragments must be assembled into a complete ciphertext. The need for implementations to support ciphertext fragmentation for decryption has been used as an attack vector several times [APW09, ADHP16, DP10].

Security models capturing symmetric encryption schemes supporting ciphertext fragmentation were first proposed by Boldyreva et al. [BDPS12]. They defined a confidentiality notion and two advanced notions: *boundary hiding* and *denial-of-service*. Their confidentiality definition was later found to contain a flaw. The flaw was repaired in [ADHP16]; that paper also added an integrity notion.

2.4.1 Syntax

Let $\mathsf{S}_\perp$ denote the set of errors that the 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. Let ¶ denote a symbol such that $\P \notin (\{0,1\} \cup \mathsf{S}_\perp)^*$. We use ¶ to indicate the end of plaintext messages. It is needed by the calling application to correctly parse the output from the decryption algorithm.

Definition 3 (Symmetric encryption scheme supporting ciphertext fragmentation)**.** A symmetric encryption scheme $\mathsf{SE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ supporting ciphertext fragmentation with associated message space $\mathcal{M} \subseteq \{0,1\}^*$, ciphertext space $\mathcal{C} \subseteq \{0,1\}^*$ and error set $\mathsf{S}_\perp$, is specified by three algorithms:

– A randomised key generation algorithm KGen that outputs a key k and initial states σ_0 and ϱ_0 for encryption and decryption, respectively. We write $(\mathsf{k}, \sigma_0, \varrho_0) \leftarrow \mathsf{KGen}$.

– A stateful encryption algorithm Enc that takes as input a key k, a plaintext $m \in \mathcal{M}$ and current state σ, and outputs a ciphertext $c \in \mathcal{C}$ and the updated state σ'. We write $(c, \sigma') \leftarrow \mathsf{Enc}_\mathsf{k}(m, \sigma)$.

– A deterministic and stateful decryption algorithm Dec that takes as input a key k, a fragment $f \in \{0,1\}^*$ and current state ϱ, and outputs a plaintext fragment $m \in (\{0, 1, \P\} \cup \mathsf{S}_\perp)^*$ and the updated state ϱ'. We write $(m, \varrho') \leftarrow \mathsf{Dec}_\mathsf{k}(f, \varrho)$.

For a vector $\mathbf{m} = (m_1, m_2, \ldots, m_r)$ we use $(\mathbf{c}, \sigma') \leftarrow \mathsf{Enc_k}(\mathbf{m}, \sigma)$ to denote $(c_1, \sigma_1) \leftarrow \mathsf{Enc_k}(m_1, \sigma)$; $(c_2, \sigma_2) \leftarrow \mathsf{Enc_k}(m_2, \sigma_1)$; $\ldots$; $(c_r, \sigma_r) \leftarrow \mathsf{Enc_k}(m_r, \sigma_{r-1})$ where $\sigma' = \sigma_r$ and $\mathbf{c} = (c_1, c_2, \ldots c_r)$. For further details we refer the reader to [BDPS12, ADHP16], in particular for a suitable correctness definition. Below, we define security properties for symmetric encryption schemes supporting ciphertext fragmentation.

2.4.2 Confidentiality and Integrity

In this work, we will use the confidentiality and integrity notions from [ADHP16]. The two notions are reproduced below.

Definition 4 (IND-sfCFA advantage)**.** Let SE = (KGen, Enc, Dec) be a symmetric encryption scheme supporting ciphertext fragmentation. Let LR and DEC be the oracles specified in Fig. 6, with both oracles being initialised according to INI as specified in Fig. 6. For any adversary $\mathcal{A}$ we define its IND-sfCFA advantage as:

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

The scheme SE is said to be (η, R)-IND-sfCFA secure if for any adversary $\mathcal{A}$ with resources at most R, its IND-sfCFA advantage $\mathsf{Adv}^{\text{ind-sfcfa}}_{\mathsf{SE}}(\mathcal{A})$ is bounded by η.

Definition 5 (IND-sfCTF advantage)**.** Let SE = (KGen, Enc, Dec) be a symmetric encryption scheme supporting ciphertext fragmentation. Let ENC and DEC be oracles specified as in Fig. 6, with both oracles being initialised according to INI as specified in Fig. 6. Let FORGE be the event that DEC returns an element from $\{0, 1, \P\}^*$. For any adversary $\mathcal{A}$, we define its IND-sfCTF advantage as:

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

The scheme SE is said to be (η, R)-IND-sfCTF secure, if for any adversary $\mathcal{A}$ with resources at most R, its IND-sfCTF advantage $\mathsf{Adv}^{\text{ind-ctf}}_{\mathsf{SE}}(\mathcal{A})$ is bounded by η.

2.4.3 Boundary Hiding

It is standard to assume that a symmetric encryption scheme is allowed to leak the length of a message. However, leaking the length of a message can be potentially dangerous [DP07, DP10, WMSM11]. The SSH BPP attempts to hide message 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 message lengths. Boldyreva et al. [BDPS12] formalised boundary hiding security notions for passive and active adversaries. These notions provide a form of length hiding for sequences of messages. They are replicated below. To save space, for passive boundary hiding (BH-CPA) the boxed code is excluded, while for active boundary hiding (BH-sfCFA) the boxed code replaces the code adjacent to it.

Definition 6 (BH-CPA $\boxed{\text{BH-sfCFA}}$ advantage)**.** Let SE = (KGen, Enc, Dec) be a symmetric encryption scheme supporting ciphertext fragmentation. Let LR-BH and DEC be the oracles specified in Fig. 6, with both oracles being initialised according to INI as specified in Fig. 6. For any adversary $\mathcal{A}$, we define its BH-CPA $\boxed{\text{BH-sfCFA}}$ advantage as:

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

The scheme SE is said to be (η, R)-BH-CPA $\boxed{(\eta, \mathsf{R})\text{-BH-sfCFA}}$ secure, if for any adversary $\mathcal{A}$ with resources at most R, its BH-CPA $\boxed{\text{BH-sfCFA}}$ advantage $\mathsf{Adv}^{\text{bh-cpa}\ \boxed{\text{bh-sfcfa}}}_{\mathsf{SE}}(\mathcal{A})$ is bounded by η.

The passive boundary hiding game captures the case where an adversary tries to gather information from ciphertext lengths. The active boundary hiding game captures an adversary that tries to infer information by means of, for example, flipping bits in the stream of fragments. 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 noted in [ADHP16], none of the schemes currently supported in OpenSSH achieves BH-sfCFA security due to simple bit-flipping attacks.

```
alg. INI
  sync ← true
  $i_e \leftarrow 0$
  $\mathsf{C} = [\,], \mathsf{M} = [\,]$
  $\mathsf{F} \leftarrow \varepsilon, \mathsf{M}' \leftarrow \varepsilon$
  $b \leftarrow_R \{0,1\}$
  $(\mathsf{k}, \sigma, \varrho) \leftarrow \mathsf{KGen}$
  return

alg. LR$(b, m_0, m_1)$
  if $|m_0| \neq |m_1|$ return $\varepsilon$
  $(c, \sigma) \leftarrow \mathsf{Enc_k}(m_b, \sigma)$
  $i_e \leftarrow i_e + 1, \mathsf{C}[i_e] \leftarrow c$
  $\mathsf{M}[i_e] \leftarrow m_b \parallel ¶$
  return $c$

alg. ENC$(m)$
  $(c, \sigma) \leftarrow \mathsf{Enc_k}(m, \sigma)$
  $i_e \leftarrow i_e + 1, \mathsf{C}[i_e] \leftarrow c$
  $\mathsf{M}[i_e] \leftarrow m \parallel ¶$
  return $c$

alg. LR-BH$(b, \mathbf{m}_0, \mathbf{m}_1)$
  $\sigma_0 \leftarrow \sigma,\ \sigma_1 \leftarrow \sigma$
  $(\mathbf{c}_0, \sigma_0) \leftarrow \mathsf{Enc_k}(\mathbf{m}_0, \sigma_0)$
  $(\mathbf{c}_1, \sigma_1) \leftarrow \mathsf{Enc_k}(\mathbf{m}_1, \sigma_1)$
  $c_0 \leftarrow \parallel(\mathbf{c}_0),\ c_1 \leftarrow \parallel(\mathbf{c}_1)$
  if $|c_0| \neq |c_1|$ return $\perp$
  $\sigma \leftarrow \sigma_b$
  for $k = 1 \ldots |c_b|$
    $i_e \leftarrow i_e + 1$
    $\mathsf{C}[k] \leftarrow \mathbf{c}_b[k]$
    $\mathsf{M}[i_e] \leftarrow \mathbf{m}_b[k]$
  return $c_b$

alg. DEC(f)
  $(m, \varrho) \leftarrow \mathsf{Dec_k}(\mathsf{f}, \varrho)$
  $\mathsf{F} \leftarrow \mathsf{F} \parallel \mathsf{f}, \mathsf{M}' \leftarrow \mathsf{M}' \parallel m$
  if sync = true
    $j_d \leftarrow \min\left(\{l \mid \parallel(\mathsf{C}[1 \ldots l]) \npreceq \mathsf{F}\} \cup \{i_e\}\right)$
    if $\mathsf{F} \preceq \parallel(\mathsf{C}[1 \ldots j_d])$
      $m \leftarrow \varepsilon$
    else
      $m \leftarrow \mathsf{M}' \% \parallel(\mathsf{M}[1 \ldots j_d - 1])$
      if $\parallel(\mathsf{C}[1 \ldots j_d]) \preceq \mathsf{F}$
        $m \leftarrow \mathsf{M}' \% \parallel(\mathsf{M}[1 \ldots j_d])$
      if $m \neq \varepsilon$
        sync ← false
  return $m$
```

Figure 6: Algorithms for defining IND-sfCFA, IND-sfCTF, BH-CPA and BH-sfCFA advantage.

2.4.4 Denial of Service

Fragmentation can aid in the successful execution of Denial-of-Service (DoS) attacks. Consider, for example, the SSH protocol and conventional encryption schemes that use encryption modes such as CBC and CTR. 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 [BDPS12] and is somewhat tailored to captured the kind of attack described above.

Definition 7 (DOS-sfCFA advantage)**.** Let SE = (KGen, Enc, Dec) be a symmetric encryption scheme supporting fragmentation. Let ENC and DEC-DOS be the oracles specified in Fig. 6 and Fig. 7 respectively, with both oracles being initialised according to INI-DOS as specified in Fig. 7. Let DENIAL be the event that WIN = 1 after a call to DEC-DOS. For any adversary $\mathcal{A}$, we define its n-DOS-sfCFA advantage as:

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

The scheme SE is said to be (η, R)-n-DOS-sfCFA secure, if for any adversary $\mathcal{A}$ with resources at most R, its n-DOS-sfCFA advantage $\mathsf{Adv}_{\mathsf{SE}}^{n\text{-}\mathsf{dos\text{-}sfcfa}}(\mathcal{A})$ 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, which 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 how long the output from the decryption algorithm can be stalled. Hence, we are interested in achieving n-DOS-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 to be equal to the maximum message size. Using the InterMAC scheme presented in Section 3 it is possible to substantially lower 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.

```
alg. INI-DOS
sync ← true
$i_e \leftarrow 0$
$C = [\,], M = [\,]$
$F \leftarrow \varepsilon, M' \leftarrow \varepsilon$
$(k, \sigma, \varrho) \leftarrow$ KGen
WIN ← 0
return
```

```
alg. DEC-DOS(f)
$(m, \varrho) \leftarrow \mathsf{Dec}_k(f, \varrho)$
$M' \leftarrow M' \,\|\, m$
if sync = true
    $F \leftarrow F \,\|\, f$
    $j_d \leftarrow \min\left(\{l \mid \|(C[1 \ldots l]) \npreceq F\} \cup \{i_e\}\right)$
    if $F \preceq \|(C[1 \ldots j_d])$
        $m \leftarrow \varepsilon$
    else
        $m \leftarrow M' \,\%\, \|(M[1 \ldots j_d - 1])$
        if $\|(C[1 \ldots j_d]) \preceq F$
            $m \leftarrow M' \,\%\, \|(M[1 \ldots j_d])$
        if $m \neq \varepsilon$
            $F \leftarrow \varepsilon$
            sync ← false
else
    if $m = \varepsilon$: $F \leftarrow F \,\|\, f$
    else: $F \leftarrow \varepsilon$
if sync = false and $|F| \geq n$
    WIN ← 1
return $m$
```

Figure 7: Algorithms for defining DOS-sfCFA advantage.

3 InterMAC

The security notions presented in Section 2.4 represent a strong set of security properties not captured by the usual security models for symmetric encryption. A symmetric encryption scheme that simultaneously meets IND-sfCFA, IND-sfCTF, BH-sfCFA and DOS-sfCFA is the InterMAC scheme from [BDPS12]. 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 symmetric encryption scheme.

We prove that the modifications made to InterMAC do not change its security properties, see Section 3.3.

3.1 Original InterMAC

Definition 8 below defines the original InterMAC scheme appearing in [BDPS12]. Note that the chunk length N, message sizes, chunk lengths, etc., are all counted in bytes compared to the original presentation. This consistent with us adopting a byte-oriented presentation of the scheme as explained in Section 2.1.

Definition 8. Let $\mathsf{SE} = (\mathsf{EGen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme with an associated message space that contains B^{N+1}, for some desired $N \in \mathbb{N}$. Without loss of generality, we assume SE is stateful. Furthermore, assume that Enc encrypts all messages of length $N+1$ bytes to ciphertexts of ℓ_{c} bytes (for some fixed ℓ_{c}). (Length-regular schemes, i.e. schemes in which any two messages of the same length always encrypt to ciphertexts of equal length, will meet this requirement.) Let $\mathsf{MA} = (\mathsf{MGen}, \mathsf{Tag}, \mathsf{Ver})$ be a message authentication code with associated message space $\{0,1\}^*$ and tag length ℓ_{tag} (counted in bytes). Then the (byte-oriented) InterMAC scheme $\mathsf{oIM} = (\overline{\mathsf{Gen}}, \overline{\mathsf{Enc}}, \overline{\mathsf{Dec}})$ defined in Fig. 8 gives a symmetric encryption scheme supporting ciphertext fragmentation with associated message space $\{\mathsf{B}^N\}^+$.

```
alg. $\overline{\mathsf{Gen}}$
$(\mathsf{k}_e, \sigma_{\mathsf{e}}, \varrho_{\mathsf{e}}) \leftarrow \mathsf{EGen}$
$\mathsf{k}_m \leftarrow \mathsf{MGen}$
$\mathsf{k} \leftarrow \mathsf{k}_e \parallel \mathsf{k}_m$
$\sigma \leftarrow (\sigma_{\mathsf{e}}, 0)$
$\varrho \leftarrow (\varrho_{\mathsf{e}}, \epsilon, \epsilon, 0, 0, 0)$
return $(\mathsf{k}, \sigma, \varrho)$

alg. $\overline{\mathsf{Enc}}(\mathsf{k}, m, (\sigma_{\mathsf{e}}, \mathsf{msg\_ctr}))$
$c \leftarrow \epsilon, b \leftarrow \mathsf{0x00}, \mathsf{k}_e \parallel \mathsf{k}_m \leftarrow \mathsf{k}$
for $\mathsf{chunk\_ctr} = 1 \ldots |m|_{\mathsf{B}}/N$
    $p \leftarrow 1 + (\mathsf{chunk\_ctr} - 1) \cdot N$
    $q \leftarrow \mathsf{chunk\_ctr} \cdot N$
    $m' \leftarrow m[p : q]_{\mathsf{B}}$
    if $q = |m|_{\mathsf{B}}$
        $b \leftarrow \mathsf{0x01}$
    $(c', \sigma_{\mathsf{e}}) \leftarrow \mathsf{Enc}_{\mathsf{k}_e}(m' \parallel b, \sigma_{\mathsf{e}})$
    $\tau \leftarrow \mathsf{Tag}_{\mathsf{k}_m}(c' \parallel \langle \mathsf{msg\_ctr} \rangle \parallel \langle \mathsf{chunk\_ctr} \rangle)$
    $c \leftarrow c \parallel c' \parallel \tau$
$\mathsf{msg\_ctr} \leftarrow \mathsf{msg\_ctr} + 1$
return $(c, (\sigma_{\mathsf{e}}, \mathsf{msg\_ctr}))$
```

```
alg. $\overline{\mathsf{Dec}}(\mathsf{k}, f, (\varrho_{\mathsf{e}}, \alpha, m, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail}))$
$\mathsf{k}_e \parallel \mathsf{k}_m \leftarrow \mathsf{k}$
$w \leftarrow \epsilon, \alpha \leftarrow \alpha \parallel f$
while $|\alpha|_{\mathsf{B}} \geq \ell_{\mathsf{c}} + \ell_{\mathsf{tag}}$
    $c \leftarrow \alpha[1 : \ell_{\mathsf{c}}]_{\mathsf{B}}$
    $\tau \leftarrow \alpha[\ell_{\mathsf{c}} + 1 : \ell_{\mathsf{c}} + \ell_{\mathsf{tag}}]_{\mathsf{B}}$
    $\alpha \leftarrow \alpha[N + 1 : |\alpha|_{\mathsf{B}}]_{\mathsf{B}}$
    $j \leftarrow \mathsf{chunk\_ctr} + 1$
    $\mathsf{v} \leftarrow \mathsf{Ver}_{\mathsf{k}_m}(c \parallel \langle \mathsf{msg\_ctr} \rangle \parallel \langle \mathsf{chunk\_ctr} \rangle, \tau)$
    if $\mathsf{v} = \perp$ and $\mathsf{fail} = 0$
        $w \leftarrow w \parallel \perp$
        $\mathsf{fail} \leftarrow 1$
    elseif $\mathsf{fail} = 1$
        $w \leftarrow w \parallel \perp$
    else
        $(m', \varrho_{\mathsf{e}}) \leftarrow \mathsf{Dec}_{\mathsf{k}_e}(c, \varrho_{\mathsf{e}})$
        $m \leftarrow m \parallel m'[1 : N]_{\mathsf{B}}$
        if $m'[N + 1]_{\mathsf{B}} \neq \mathsf{0x00}$
            $w \leftarrow w \parallel m \parallel \P$
            $\mathsf{msg\_ctr} \leftarrow \mathsf{msg\_ctr} + 1 : \mathsf{chunk\_ctr} \leftarrow 0$
            $m \leftarrow \epsilon$
return $(w, (\varrho_{\mathsf{e}}, \alpha, m, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail}))$
```

Figure 8: A byte-oriented version of the original InterMAC scheme, oIM.

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 $c_1, c_2, \ldots$, called ciphertext chunks. A MAC is computed over each chunk c_i, together with the message counter $\mathsf{msg\_ctr}$ and the chunk counter $\mathsf{chunk\_ctr}$, producing a MAC tag τ_i. Finally, all ciphertext chunks and associated MAC tags are concatenated, which yields the final ciphertext $c = c_1 \parallel \tau_1 \parallel c_2 \parallel \tau_2 \parallel \cdots$. When decrypting, the fragment f is appended to the buffer α. If the buffer contains more than $\ell_{\mathsf{c}} + \ell_{\mathsf{tag}}$ bytes, the ciphertext chunk c and MAC tag τ are extracted, and the extracted data is removed from the buffer α. The MAC tag is verified over the ciphertext chunk c, 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 w. If the MAC verification passes, the ciphertext chunk is decrypted into a message m' and is appended to the current plaintext buffer m. If the final ciphertext chunk is decrypted the output buffer w is appended with the current plaintext buffer m 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. [BDPS12] proved that the original InterMAC scheme oIM is IND-sfCFA, BH-sfCFA and $(\ell_c + \ell_{tag})$-DOS-sfCFA secure. They did not prove that oIM is IND-sfCTF secure but this can be proven with little effort. The requirements on the internal symmetric encryption scheme SE and the internal MAC scheme MA are standard: SE must be IND\$-CPA secure and length-regular, while MA must be UF-CMA secure and $\mathcal{T}$ a PRF. For formal definitions of these notions, see [BDPS12].

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]

3.2 Modified InterMAC

The oIM scheme described in Section 3.1 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 add_padding$(\cdot,\cdot)$ in Figure 9 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 10, while the practical effects of padding on bandwidth and speed are explored in detail in Section 5.

[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.

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 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 remove_padding function in Figure 10 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 5.

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 msg_ctr and chunk counter 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 3.3. 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 4.1. Thirdly, using nonce-based AE makes the presentation of InterMAC cleaner.

The formal definition of the IM scheme follows.

Definition 9. Let $\mathsf{nAE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a nonce-based AE symmetric encryption scheme with an associated message space that contains B^{N+1}, for some desired $N \in \mathbb{N}$ and that has nonce space $\{0,1\}^n$. Choose $a, b \in \mathbb{N}$ such that $a + b = n$. Furthermore, assume that nAE 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 Fig. 9 gives a symmetric encryption scheme supporting ciphertext fragmentation with associated message space B^*.

3.3 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 [BDPS12]. However, the security model turned out to be buggy [ADHP16], 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 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 $\mathsf{f}_1, \mathsf{f}_2, \ldots, \mathsf{f}_v$, 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 F_s denote the (byte) string $\mathsf{f}_1 \parallel \mathsf{f}_2 \parallel \cdots \parallel \mathsf{f}_s$. F_s is

```
alg. $\overline{\mathsf{Gen}}$
  $\mathsf{k} \leftarrow \mathsf{Gen}$
  $\sigma \leftarrow (0)$
  $\varrho \leftarrow (\epsilon, \epsilon, 0, 0, 0)$
  return $(\mathsf{k}, \sigma, \varrho)$

alg. $\overline{\mathsf{Enc}}(\mathsf{k}, m, \mathsf{msg\_ctr})$
  $c \leftarrow \epsilon$
  $(m, d) \leftarrow \mathsf{add\_padding}(m, N)$
  for $\mathsf{chunk\_ctr} = 1 \ldots |m|_{\mathsf{B}}/N$
    $p \leftarrow 1 + (\mathsf{chunk\_ctr} - 1) \cdot N$
    $q \leftarrow \mathsf{chunk\_ctr} \cdot N$
    $m' \leftarrow m[p : q]_{\mathsf{B}}$
    $\mathsf{nonce} \leftarrow \langle \mathsf{msg\_ctr} \rangle_a \parallel \langle \mathsf{chunk\_ctr} \rangle_b$
    if $q = |m|_{\mathsf{B}}$
      $c' \leftarrow \mathsf{Enc_k}(m' \parallel d, \mathsf{nonce})$
    else
      $c' \leftarrow \mathsf{Enc_k}(m' \parallel \mathsf{0x00}, \mathsf{nonce})$
    $c \leftarrow c \parallel c'$
  $\mathsf{msg\_ctr} \leftarrow \mathsf{msg\_ctr} + 1$
  return $(c, (\mathsf{msg\_ctr}))$
```

```
alg. $\overline{\mathsf{Dec}}(\mathsf{k}, f, (\alpha, m, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail}))$
  $w \leftarrow \epsilon$
  $\alpha \leftarrow \alpha \parallel f$
  while $|\alpha|_{\mathsf{B}} \geq \ell_{\mathsf{c}}$
    $c \leftarrow \alpha[1 : \ell_{\mathsf{c}}]_{\mathsf{B}}$
    $\alpha \leftarrow \alpha[\ell_{\mathsf{c}} + 1 : |\alpha|_{\mathsf{B}}]_{\mathsf{B}}$
    $\mathsf{chunk\_ctr} \leftarrow \mathsf{chunk\_ctr} + 1$
    $\mathsf{nonce} \leftarrow \langle \mathsf{msg\_ctr} \rangle_a \parallel \langle \mathsf{chunk\_ctr} \rangle_b$
    $m' \leftarrow \mathsf{Dec_k}(c, \mathsf{nonce})$
    if $m' = \perp$ and $\mathsf{fail} = 0$
      $w \leftarrow w \parallel \perp$
      $\mathsf{fail} \leftarrow 1$
    elseif $\mathsf{fail} = 1$
      $w \leftarrow w \parallel \perp$
    else
      $\mathsf{chunk\_del} \leftarrow m'[N + 1]_{\mathsf{B}}$
      $m' \leftarrow \mathsf{remove\_padding}(m'[1 : N]_{\mathsf{B}}, N, \mathsf{chunk\_del})$
      if $\mathsf{chunk\_del} \neq \mathsf{0x00}$
        $w \leftarrow w \parallel m \parallel m' \parallel \P$
        $\mathsf{msg\_ctr} \leftarrow \mathsf{msg\_ctr} + 1, \mathsf{chunk\_ctr} \leftarrow 0$
        $m \leftarrow \epsilon$
      else
        $m \leftarrow m \parallel m'[1 : N]_{\mathsf{B}}$
  return $(w, (\alpha, m, \mathsf{msg\_ctr}, \mathsf{chunk\_ctr}, \mathsf{fail}))$
```

Figure 9: The modified InterMAC scheme IM; functions add_padding and remove_padding are defined in Figure 10.

```
alg. $\mathsf{add\_padding}(m, N)$
  $\mathsf{size} \leftarrow |m|_{\mathsf{B}}$
  $\mathsf{mod} \leftarrow \mathsf{size} \mod N$
  if $\mathsf{mod} = 0$
    return $(m, \mathsf{0x01})$
  $\mathsf{padlen} \leftarrow N - \mathsf{mod}$
  if $m[\mathsf{size}]_{\mathsf{B}} = \mathsf{0x00}$
    $\mathsf{padbyte} \leftarrow \mathsf{0x01}$
  else
    $\mathsf{padbyte} \leftarrow \mathsf{0x00}$
  repeat padlen times
    $m \leftarrow m \parallel \mathsf{padbyte}$
  return $(m, \mathsf{0x02})$
```

```
alg. $\mathsf{remove\_padding}(m, N, \mathsf{chunk\_del})$
  $\mathsf{padbyte} \leftarrow m[N]_{\mathsf{B}}$
  $\mathsf{padlen} \leftarrow 0, \mathsf{flag} \leftarrow 0$
  for $i = 1, \ldots N - 1$
    $\mathsf{flag} \leftarrow \mathsf{flag} \mid (m[N - j]_{\mathsf{B}} \oplus \mathsf{padbyte})$
    $\mathsf{lsb} \leftarrow (\mathsf{flag} \mid (-\mathsf{flag})) \gg 7$
    $\mathsf{add} \leftarrow \mathsf{lsb} \oplus \mathsf{0x01}$
    $\mathsf{padlen} \leftarrow \mathsf{padlen} + \mathsf{add}$
  $\mathsf{mult} \leftarrow \mathsf{chunk\_del} \cdot (\mathsf{chunk\_del} - 1) \gg 1$
  $\mathsf{padlen} \leftarrow \mathsf{padlen} \cdot \mathsf{mult}$
  $m \leftarrow m[1, N - \mathsf{padlen}]_{\mathsf{B}}$
  return $m$
```

Figure 10: 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.

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 $\|(\mathsf{C}[1 \ldots s])$, where C as is defined in Figure 6. We let i_e denote the specific value of the variable of the same name in Figure 6 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 queries, that is, up to the point in time where f_v is queried.

Assume that f_v is queried to the decryption query. There are exactly two sets of conditions on C and F for which the sync flag can be set to false during the processing of f_v. The two sets of conditions are:

A. $\underline{|\mathsf{F}_v|_\mathsf{B} > |\mathsf{CS}_{i_e}|_\mathsf{B} \text{ and } \mathsf{CS}_{i_e} \preceq \mathsf{F}_v}$. This implies $|\mathsf{F}_{v-1}|_\mathsf{B} \leq |\mathsf{CS}_{i_e}|_\mathsf{B}$ and $\mathsf{F}_{v-1} \preceq \mathsf{CS}_{i_e}$ because v is minimal. Furthermore, we must have $j_d = i_e$ since $\mathsf{CS} \preceq \mathsf{F}_v$. Since IM does not output anything before processing at least ℓ_c bytes, we can find an integer λ such that

$$\begin{aligned}
&\mathsf{F}[\lambda \cdot \ell_\mathsf{c} + 1 : (\lambda+1) \cdot \ell_\mathsf{c}] \npreceq (\mathsf{CS}i_e \,\%\, \mathsf{F}_{v-1}),\\
&\mathsf{F}[\lambda \cdot \ell_\mathsf{c} + 1 : (\lambda+1) \cdot \ell_\mathsf{c}] \preceq (\mathsf{F}_v \,\%\, \mathsf{F}_{v-1}),\\
&\mathsf{F}[\lambda \cdot \ell_\mathsf{c} + 1 : (\lambda+1) \cdot \ell_\mathsf{c}] \leftarrow \alpha[1 : \ell_\mathsf{c}]_\mathsf{B}.
\end{aligned}$$

Assume λ is minimal and set $\delta = \mathsf{F}[\lambda \cdot \ell_\mathsf{c} + 1 : (\lambda+1) \cdot \ell_\mathsf{c}]$. The assignment in the third line above happens at some point during the execution of the "while" loop in the decryption function of IM and implies that, at some point, δ is an input to Dec.

B. $\underline{\text{There exists an integer } \mu \text{ such that } \mathsf{F}_v[\mu]_\mathsf{B} \neq \mathsf{CS}_{i_e}[\mu]_\mathsf{B}}$. Assume μ is minimal. Let $t \leq i_e$ and λ be the uniqe integers satisfying:

$$|\mathsf{CS}_{t-1}|_\mathsf{B} < \lambda \cdot \ell_\mathsf{c} + 1 \leq \mu \leq (\lambda+1) \cdot \ell_\mathsf{c} \leq |\mathsf{CS}_t|_\mathsf{B}.$$

We must have $j_d = t$ and therefore $\mathsf{CS}_{j_d} \npreceq \mathsf{F}_v$. Set $\delta = \mathsf{F}[\lambda \cdot \ell_\mathsf{c} + 1 : (\lambda+1) \cdot \ell_\mathsf{c}]$.

In both cases, δ is an input to the decryption function of the underlying nonce-based encryption scheme Dec. Furthermore, all plaintext, decrypted from in-sync ciphertext, is removed from the output buffer m, and the output from the call $\mathsf{Dec}_k(\delta, \mathsf{nonce})$ is recorded in m (since $m \neq \varepsilon$). Note that we know the exact position at which the output from $\mathsf{Dec}_k(\delta, \mathsf{nonce})$ appears in m. Denote this position by m_δ. Using δ we define the following event:

$$\mathsf{bad}\text{: } \mathsf{Dec} \text{ does not return } \perp \text{ on input } \delta.$$

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 1 (IM is ℓ_c-DOS-sfCFA secure)**.** *Let* IM *be instantiated with the nonce-based enryption scheme* $\mathsf{nSE} = (\mathsf{K}, \mathsf{Enc}, \mathsf{Dec})$*. For any adversary* $\mathcal{A}_\mathsf{dos}$*, respecting restrictions* R1 *and* R2*, against* IM*, there exists an* nAE *adversary* $\mathcal{A}_\mathsf{nae}$ *against* nSE *such that:*

$$\mathsf{Adv}^{\ell_\mathsf{c}\text{-dos-sfcfa}}_{\mathsf{SE}}(\mathcal{A}_\mathsf{dos}) \leq \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}_\mathsf{nae}), \tag{1}$$

where $\mathcal{A}_\mathsf{nae}$ *uses resources similar to* $\mathcal{A}_\mathsf{dos}$*.*

Proof. It is always possible for an adversary to win the n-$\mathsf{DOS\text{-}sfCFA}$ 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:

$$\mathsf{Adv}^{n\text{-}\mathsf{dos\text{-}sfcfa}}_{\mathsf{SE}}(\mathcal{A}) \leq \Pr[\mathsf{bad}]\,. \tag{2}$$

We next show that the probability of the event bad is bounded by the probability of winning the nAE game. Let $(\mathcal{O}_1, \mathcal{O}_2)$ be oracles such that $(\mathcal{O}_1, \mathcal{O}_2) \in \{(\mathsf{ENC}, \mathsf{DEC}), (\$, \perp)\}$ 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 (nAE) INI and thereafter run $\mathcal{A}_\mathsf{dos}$, answering its queries to ENC and $\mathsf{DEC\text{-}DOS}$ as specified below. While answering queries, $\mathcal{A}_\mathsf{nae}$ maintains the lists: M, M', C and F. When the flag sync is set to false, $\mathcal{A}_\mathsf{nae}$ can detect if bad occurs and if this happens, $\mathcal{A}_\mathsf{nae}$ outputs 1, otherwise outputs 0.

$\mathsf{ENC}(\cdot)$: On input m, simulate ENC and replace the call to Enc with the oracle $\mathcal{O}_1$.

$\mathsf{DEC\text{-}DOS}(\cdot)$: On input f, simulate $\mathsf{DEC\text{-}DOS}$ and replace the call to Dec with the oracle $\mathcal{O}_2$.

The simulation of ENC and $\mathsf{DEC\text{-}DOS}$ is perfect up to the point in time where the sync flag is set to false. If the event bad occurs, we can check the position m_δ. If the position contains $\perp$ (or nothing) then we know that $(\mathcal{O}_1, \mathcal{O}_2) = (\$, \perp)$. But if the position does not contain $\perp$ then we know that $(\mathcal{O}_1, \mathcal{O}_2) = (\mathsf{ENC}, \mathsf{DEC})$. By the output from $\mathcal{A}_\mathsf{nae}$ we have:

$$\Pr[\mathsf{bad}] \leq \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}_\mathsf{nae}). \tag{3}$$

Combining (2) and (3) yields (1). It can be checked that $\mathcal{A}_\mathsf{nae}$ uses the same resources as $\mathcal{A}_\mathsf{dos}$. □

We proceed to show that IM also provides active boundary hiding.

Theorem 2 (IM is $\mathsf{BH\text{-}sfCFA}$ secure)**.** *Let* IM *be instantiated with the nonce-based enryption scheme* $\mathsf{nSE} = (\mathsf{K}, \mathsf{Enc}, \mathsf{Dec})$*. For any adversary* $\mathcal{A}_\mathsf{bh}$*, respecting restrictions* $\mathsf{R1}$ *and* $\mathsf{R2}$*, against* IM*, there exists an* nAE *adversary* $\mathcal{A}_\mathsf{nae}$ *against* nSE *such that:*

$$\mathsf{Adv}^{\mathsf{bh\text{-}sfcfa}}_{\mathsf{SE}}(\mathcal{A}_\mathsf{bh}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}_\mathsf{nae}), \tag{4}$$

where $\mathcal{A}_\mathsf{nae}$ *uses resources similar to* $\mathcal{A}_\mathsf{bh}$*.*

Proof. We prove the theorem through a sequences of games. For each game $\mathsf{G}s$, let WIN_s represent the event that the adversary computes the bit b correctly in that game.

G0 This is the $\mathsf{BH\text{-}sfCFA}$ game instantiated with nSE.

G1 In this game, we modify the decryption oracle. When the 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 the number of chunks contained in m after the remainder between M' and (possibly a substring of) M has been taken. 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 $\overline{\mathsf{Dec}}$.

This number can be computed by looking at the difference between the first member in the state variable output from $\overline{\mathsf{Dec}}$, and the decryption oracle query f. The games G0 and G1 are identical up 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. Using the same arguments as in the proof of Theorem 1, we can construct an adversary $\mathcal{A}1_{\mathsf{nae}}$ such that:

$$\Pr[\mathsf{G0} : \mathsf{WIN}_0] - \Pr[\mathsf{G1} : \mathsf{WIN}_1] \leq \Pr[\mathsf{bad}] \leq \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}1_{\mathsf{nae}}). \tag{5}$$

G2 In this game, we modify the decryption oracle to use the lists M, C, and 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 thefore does not change the output distribution. Therefore:

$$\Pr[\mathsf{G1} : \mathsf{WIN}_1] = \Pr[\mathsf{G2} : \mathsf{WIN}_2]. \tag{6}$$

G3 In this game, we replace calls to Enc with calls to \$ and replace calls to Dec with calls to $\perp$. Let $\mathcal{A}2_{\mathsf{nae}}$ be an adversary having acces to oracles $(\mathcal{O}_1, \mathcal{O}_2) \in \{(\mathsf{ENC}, \mathsf{DEC}), (\$, \perp)\}$. Define $\mathcal{A}2_{\mathsf{nae}}$ as follows: $\mathcal{A}2_{\mathsf{nae}}$ runs (nAE) INI and thereafter runs $\mathcal{A}_{\mathsf{bh}}$ using oracles $(\mathcal{O}_1, \mathcal{O}_2)$ to simulate the encryption and decryption oracles. If $\mathcal{A}_{\mathsf{bh}}$ 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 G2 and if WIN_2 occurs, $\mathcal{A}2_{\mathsf{nae}}$ outputs 1. If $(\mathcal{O}_1, \mathcal{O}_2) = (\$, \perp)$, $\mathcal{A}2_{\mathsf{nae}}$ perfectly simulates game G3 and if WIN_3 occurs, $\mathcal{A}2_{\mathsf{nae}}$ outputs 1. Therefore:

$$\begin{aligned}\Pr[\mathsf{G2} : \mathsf{WIN}_2] - \Pr[\mathsf{G3} : \mathsf{WIN}_3] &= \Pr\left[\mathsf{INI} : \mathcal{A}2_{\mathsf{nae}}^{\mathsf{ENC}(\cdot,\cdot),\mathsf{DEC}(\cdot,\cdot)} = 1\right] \\ &\quad - \Pr\left[\mathsf{INI} : \mathcal{A}2_{\mathsf{nae}}^{\$(\cdot,\cdot),\perp(\cdot,\cdot)} = 1\right] \\ &\leq \mathsf{Adv}^{\mathsf{nAE}}_{\mathsf{nSE}}(\mathcal{A}2_{\mathsf{nae}}). \end{aligned} \tag{7}$$

Consider the game 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 output by the oracle \$. The adversary therefore does not learn anything from the combination of decryption and encryption queries. Hence:

$$\Pr[\mathsf{G3} : \mathsf{WIN}_3] = \frac{1}{2}. \tag{8}$$

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 (5), (6), (7) and (8) yields (4). It can be checked that $\mathcal{A}_{\mathsf{nae}}$ uses resources similar to $\mathcal{A}_{\mathsf{bh}}$. □

Theorem 5.3 of [BDPS12] showed that BH-sfCFA security implies IND-sfCFA security and was used to prove that oIM is IND-sfCFA secure. However, since we have followed [ADHP16] in correcting the security models, this result needs to be re-established. Luckily, as we show below, we can recover the result of [BDPS12, Theorem 5.3] for our modified security models. As a consequence, IM is IND-sfCFA secure (under restrictions R1 and R2).

Theorem 3 (BH-sfCFA implies IND-sfCFA)**.** *Let* $\mathsf{nSE} = (\mathsf{K}, \mathsf{Enc}, \mathsf{Dec})$ *be a length-regular nonce-based encryption scheme. For any* IND-sfCFA *adversary* $\mathcal{A}_{\mathsf{cfa}}$*, there exists a* BH-sfCFA *adversary* $\mathcal{A}_{\mathsf{bh}}$ *such that:*

$$\mathsf{Adv}^{\mathsf{ind\text{-}sfcfa}}_{\mathsf{SE}}(\mathcal{A}_{\mathsf{cfa}}) \leq \mathsf{Adv}^{\mathsf{bh\text{-}sfcfa}}_{\mathsf{SE}}(\mathcal{A}_{\mathsf{bh}}), \tag{9}$$

where $\mathcal{A}_{\mathsf{bh}}$ *uses resources similar to* $\mathcal{A}_{\mathsf{cfa}}$*.*

Proof. The proof is straightforward. For any adversary $\mathcal{A}_{\mathsf{cfa}}$, we construct $\mathcal{A}_{\mathsf{bh}}$ as follows:

$\mathcal{A}_{\mathsf{bh}}$: Run $\mathcal{A}_{\mathsf{cfa}}$. Forward any encryption queries from $\mathcal{A}_{\mathsf{cfa}}$ to the encryption oracle of $\mathcal{A}_{\mathsf{bh}}$ after checking that the lengths of the two plaintexts are the same. Since nSE is length-regular, the outputs from LR and $\mathsf{LR\text{-}BH}$ are consistent. Likewise, $\mathcal{A}_{\mathsf{bh}}$ forwards any decryption queries from $\mathcal{A}_{\mathsf{cfa}}$ to its own decryption oracle. $\mathcal{A}_{\mathsf{bh}}$ forwards the output returned on its queries to $\mathcal{A}_{\mathsf{cfa}}$. The guess b' by $\mathcal{A}_{\mathsf{cfa}}$ is also the guess by $\mathcal{A}_{\mathsf{bh}}$.

The simulation of LR and DEC by $\mathcal{A}_{\mathsf{bh}}$ is perfect. Hence:

$$\mathsf{Adv}_{\mathsf{SE}}^{\mathsf{ind\text{-}sfcfa}}(\mathcal{A}_{\mathsf{cfa}}) = \mathsf{Adv}_{\mathsf{SE}}^{\mathsf{bh\text{-}sfcfa}}(\mathcal{A}_{\mathsf{bh}}),$$

which implies (9). It can be checked that $\mathcal{A}_{\mathsf{bh}}$ uses the same resources as $\mathcal{A}_{\mathsf{cfa}}$. □

Finally, we prove that IM is $\mathsf{IND\text{-}sfCTF}$ secure. Because this definition first appeared in [ADHP16] the original InterMAC scheme oIM was not proved $\mathsf{IND\text{-}sfCTF}$ secure in [BDPS12]. However, only minor modifications in the proof below are required to also prove that oIM is $\mathsf{IND\text{-}sfCTF}$ secure.

Theorem 4 (IM is $\mathsf{IND\text{-}sfCTF}$ secure)**.** *Let* IM *be instantiated with the nonce-based enryption scheme* $\mathsf{nSE} = (\mathsf{K}, \mathsf{Enc}, \mathsf{Dec})$. *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:*

$$\mathsf{Adv}_{\mathsf{SE}}^{\mathsf{ind\text{-}ctf}}(\mathcal{A}_{\mathsf{ctf}}) \leq \mathsf{Adv}_{\mathsf{nSE}}^{\mathsf{nAE}}(\mathcal{A}_{\mathsf{nae}}), \tag{10}$$

where $\mathcal{A}_{\mathsf{nae}}$ *uses resources similar to* $\mathcal{A}_{\mathsf{ctf}}$.

Proof. We prove the theorem through a sequence of games.

G0 This is the $\mathsf{IND\text{-}sfCTF}$ game instantiated with nSE.

G1 In this game, we make modifications identical to the changes made in the game $\mathsf{G1}$ in the proof of Theorem 2. Using an identical argument, we can construct an adversary $\mathcal{A}_{\mathsf{nae}}$ such that:

$$\Pr[\mathsf{G0} : \mathsf{FORGE}] - \Pr[\mathsf{G1} : \mathsf{FORGE}] \leq \Pr[\mathsf{bad}] \leq \mathsf{Adv}_{\mathsf{nSE}}^{\mathsf{nAE}}(\mathcal{A}_{\mathsf{nae}}). \tag{11}$$

Observe that the decryption oracle in game $\mathsf{G1}$ will never output anything from the set $\{0, 1, \P\}^*$. Hence, the event FORGE will never occur in game $\mathsf{G1}$. Therefore:

$$\Pr[\mathsf{G1} : \mathsf{FORGE}] = 0. \tag{12}$$

Combining (11) and (12) yields (10). It can be checked that $\mathcal{A}_{\mathsf{nae}}$ uses the same resources as $\mathcal{A}_{\mathsf{ctf}}$.

□

This concludes our security analysis of IM.

4 libInterMAC

This section describes libInterMAC, a reference C-implementation of the IM scheme presented in Section 3.2. We also touch on some of the challenges involved in bringing InterMAC into practice. Especially, we discuss side-channels and possible mitigations, cf. Section 4.6.2. libInterMAC is available at [AHP18a].

4.1 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. The active boundary hiding proved especially difficult and is discussed in detail in Section 4.6.2.

4.1.1 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 [BLS12, ABF$^+$17]. Consider, for example, software that implements the nonce-based AE scheme AES-GCM and allows the user to specify the nonce for each invocation of the encryption function. If the user at any point reuses a nonce, under the same encryption key, part of the key is leaked [Jou06, BZD$^+$16].

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.

4.1.2 Extensibility

libInterMAC defines an interface to represent the nonce-based AE scheme required in the IM construction and utilise 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 [Hou15] 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.

4.2 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.

4.3 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 9 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 functions are applied to each chunk. Therefore, the nonce will not repeat before 2^{64} messages have been processed or divide into more than 2^{32} chunks. Consequently, for a fixed key, libInterMAC can encrypt

```
alg. ChaCha20-Poly1305-enc(k, nonce, $m$)
1 : block_ctr ← 0
2 : $k_{poly}$ ← ChaCha20(k, $0x00^{32}$, nonce, block_ctr)
3 : block_ctr ← 1
4 : $\tau$ ← Poly1305($k_{poly}$, $c$)
5 : $c$ ← ChaCha20(k, $m$, nonce, block_ctr)
6 : $\tau$ ← Poly1305($k_{poly}$, $c$)
7 : return $c \| \tau$
```

```
alg. ChaCha20-Poly1305-dec(k, nonce, $c$, $\tau_{in}$)
1 : block_ctr ← 0
2 : $k_{poly}$ ← ChaCha20(k, $0x00^{32}$, nonce, block_ctr)
3 : $\tau$ ← Poly1305($k_{poly}$, $c$)
4 : if $\tau \neq \tau_{in}$: return $\perp$
5 : block_ctr ← 1
6 : $m$ ← ChaCha20(k, $c$, nonce, block_ctr)
7 : return $m$
```

Figure 11: The nonce-based AE scheme ChaCha20-Poly1305.

a maximum of 2^{64} messages of 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 4.4 discuss specific restrictions for each supported nonce-based AE scheme.

4.4 Supported Nonce-based AE Schemes

We have implemented support for two different nonce-based AE schemes: AES-GCM and ChaCha20-Poly1305. Below we describe how each scheme is implemented, and how they consume the nonce generated in libInterMAC.

4.4.1 ChaCha20-Poly1305

ChaCha20 is the stream cipher defined in [Ber08] and Poly1305 is the one-time MAC defined in [NL15]. To describe the ChaCha20-Poly1305 implementation in libInterMAC we define the following interfaces for ChaCha20 and Poly1305: ChaCha20 takes a 32-byte key k, a variable-length plaintext m, a 12-byte nonce nonce and a 4-byte initial block counter block_ctr and outputs an encryption c of m. We write $c \leftarrow$ ChaCha20(k, m, nonce, block_ctr). Reversing the roles of m and c yields the corresponding decryption process. Poly1305 takes a 32-byte key k and a variable-length string str and outputs a 16-byte tag τ. We write $\tau \leftarrow$ Poly1305(k, str).

The nonce-based AE scheme ChaCha20-Poly1305 implemented in libInterMAC closely follows the AEAD composition of ChaCha20 and Poly1305 defined in RFC 7539 [NL15] and is depicted in Fig. 11. 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 Poly1305 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 $\|$ c $\|$ τ 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 11, 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.

4.4.2 AES-GCM

The AES-GCM AE scheme is specified in [MV04]. In libInterMAC, the AES-GCM AE scheme AES-GCM is implemented using the Libcrypto EVP API with 128-bit AES. Libcrypto is part of the OpenSSL Toolkit [Com18] 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 dramatically increases the performance of AES-GCM.

4.4.3 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 instructions (`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 be found in Section 4.7. A comparison for IM when used with AES-GCM and ChaCha20-Poly1305 to implement SSH cipher suites can be found in Section 5.3.

4.5 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 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 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 function invocations, a limit on the allowed number of encrypted chunks when using AES-GCM, is also enforced, see Table 2 (bottom). There are no restrictions on the number of encrypted chunks for ChaCha20-Poly1305.

Table 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.

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 2: 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.

chunk length **success prob.**	2^7	2^8	2^9	2^{10}	$\cdots$	2^k
2^{-60}	$2^{31.5}$	$2^{30.5}$	$2^{29.5}$	$2^{28.5}$	$\cdots$	$2^{38.5-k}$
2^{-50}	$2^{41.5}$	$2^{40.5}$	$2^{39.5}$	$2^{38.5}$	$\cdots$	$2^{48.5-k}$
2^{-40}	$2^{51.5}$	$2^{50.5}$	$2^{49.5}$	$2^{48.5}$	$\cdots$	$2^{58.5-k}$
libInterMAC **limit on # chunks**	2^{41}	2^{40}	2^{39}	2^{38}	$\cdots$	2^{48-k}

4.5.1 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 Poly1305 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 attempt [Ber05].

[Pro14] shows that that security degradation derived for ChaCha20 and Poly1305 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 Poly1305 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.

4.5.2 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 $1 - 2^{128-k}$ [Fer05].

From [LP17], 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 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 1 and Table 2, respectively.

4.6 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 deployed schemes in OpenSSH). When also considering side-channels as a possible attack vector, it becomes paramount that the execution time of the decryption function 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 [Res18, Appendix E.3].

4.6.1 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 [Res18, 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 10 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 9, 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 17. Moreover, with our pseudo-code for IM as written in Figure 9, 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 18-20 in Figure 9) 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 4.7.

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.

4.6.2 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 $|\text{decrypted chunk}|$ 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.

4.6.3 Active Boundary Hiding in Practice

In some types of application, it is *not* possible to achieve the active boundary hiding notion BH-sfCFA in practice. For example, 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 IM, an 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*.

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 security of the IM scheme.

The above discussion highlights a discrepancy between theory and practice concerning the boundary hiding security notions defined in Section 2.4. 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 is 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.

4.7 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 3 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 $N = B - 1 \mod B$, 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 scheme. We focus on comparing the 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 12. 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 13.

Choosing the chunk length to be $N = 0 \mod 16$ (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 $N = 15 \mod 16$ (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 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 12. As the chunk length grows and the ratio between the message length and the chunk length becomes smaller, the cost between choosing $N = 0 \mod 16$ and $N = 15 \mod 16$ evens out and, eventually, shifts in favour of the former.

Figure 13 indicates that choosing the chunk length $N = 0 \mod 16$ is better when the chunk length is close to the message length and that the cross-over (the point at which $N = 15 \mod 16$ becomes the better choice) happens when the message length is significantly larger than the chunk length. These observations agree with the observations from Figure 12.

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 5.3.

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 1 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 2 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

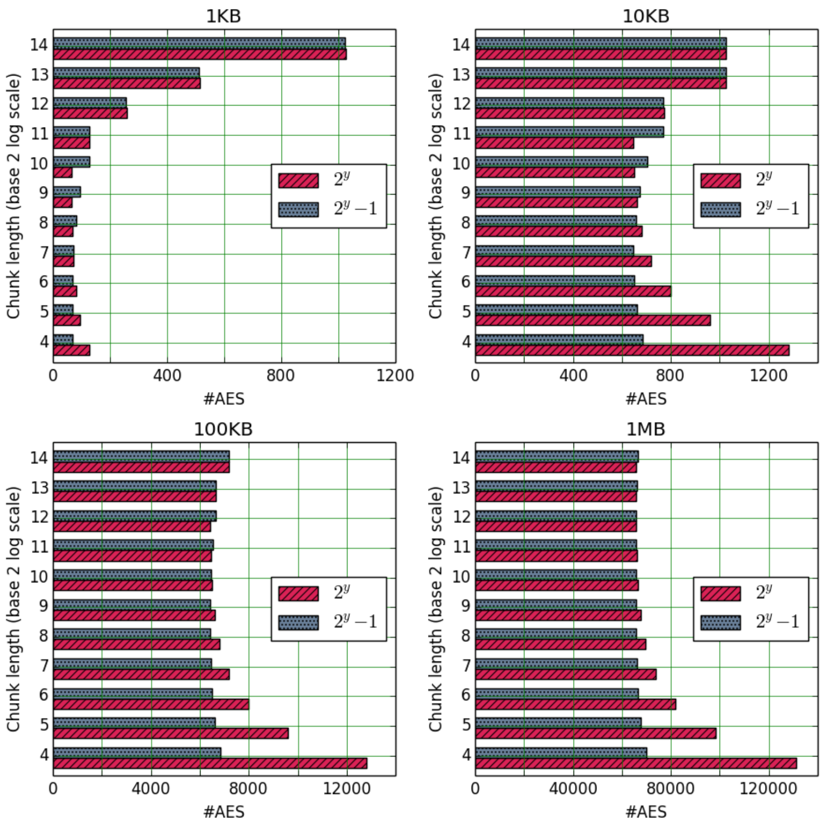

Figure 12: Number of AES operations needed to IM-encrypt a message as a function of the chunk length, with the choice of AES-GCM as internal nonce-based AE scheme. Plots are given for 4 different message lengths: 1KB, 10KB, 100KB and 1MB.

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 14, which compares this option with decryption using simple non-constant-time padding removal code.

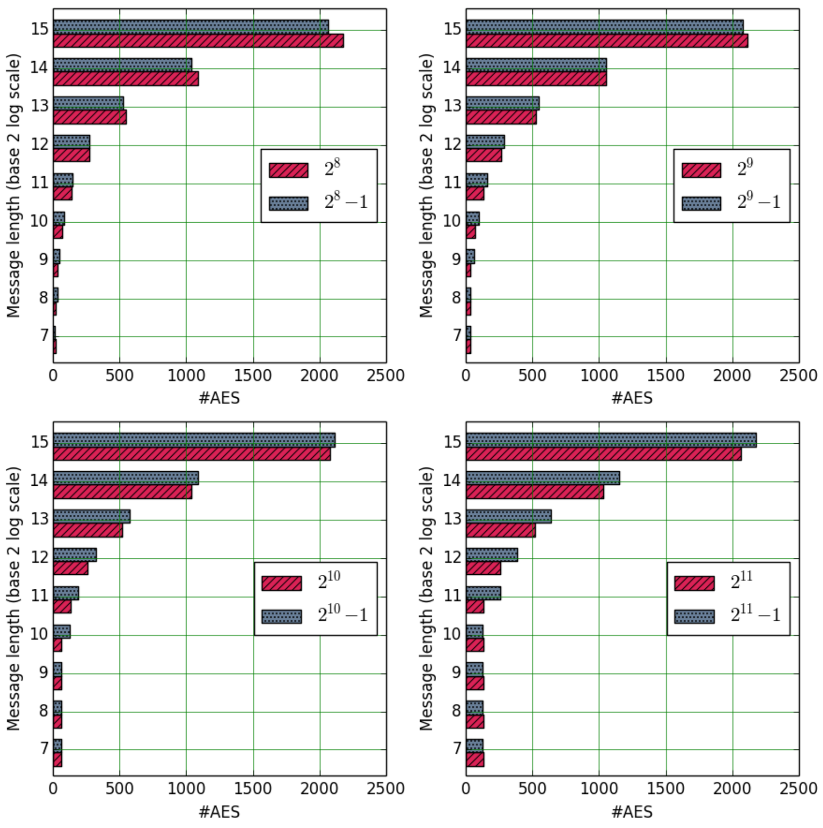

Figure 13: Number of AES operations needed to IM-encrypt a message as a function of the message length, with the choice of AES-GCM as internal nonce-based AE scheme. Plots are given for 8 different chunk lengths: 2^8-1, 2^8, 2^9-1, 2^9, $2^{10}-1$, 2^{10}, $2^{11}-1$ and 2^{11}.

5 Case Study: IM-based Schemes in OpenSSH

In this section, we extend the widely-deployed SSH implementation OpenSSH with InterMAC-based schemes. In the following, we will denote this extension of OpenSSH as IMOpenSSH. The IM-based schemes are obtained using libInterMAC. Using IMOpenSSH is as easy as using existing OpenSSH schemes and requires no special knowledge about IM apart from making a choice of the chunk length and the internal nonce-based AE scheme. IMOpenSSH is available at [AHP18b].

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.

5.1 SSH Encryption Scope

Data in SSH is sent in SSH packets defined in the Binary Packet Protocol (BPP) described in RFC 4253 [YL06]. SSH Packets are constructed in a two-step process: data encoding and cryptographic processing.

The encoding is performed 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 padding appended (consisting of random bytes). The length field has 4 bytes and encodes the combined length in bytes of the padding

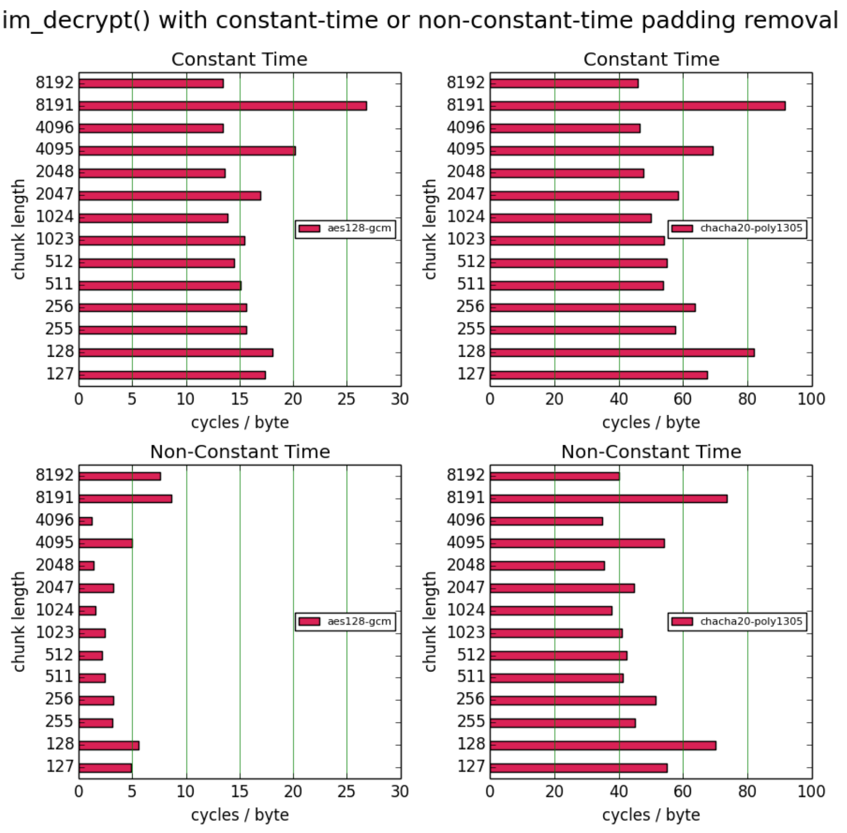

Figure 14: Comparing the cost of decryption using constant-time and non-constant-time padding removal in libInterMAC, with AES-GCM and ChaCha20-Poly1305 as internal nonce-based AE schemes. The constant-time option involves a significant performance penalty, especially for AES-GCM.

length field, payload and padding. The padding length field has 1 byte and encodes the length in bytes of the padding. The 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 (including 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.

SSH provides confidentiality and integrity through symmetric encryption and message authentication codes. 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 plaintext packet, enforcing an Encrypt-and-MAC paradigm.

When the AE schemes AES-GCM (a modified version of [IS09], see [FPR+13]) and Generic Encrypt-then-MAC (as described in [ADHP16]) were made available as encryption schemes in OpenSSH, the encryption scope had to be modified. The reason is that the BPP specifies that the length field must be encrypted, but the OpenSSH AES-GCM and Generic Encrypt-then-MAC schemes for OpenSSH also require that a MAC tag be verified before any decryption occurs. Both conditions cannot be met simultaneously because the length field specifies the location of the MAC tag in the incoming data stream. The solution chosen by OpenSSH was to exclude the length field from the encryption scope. An immediate consequence of this decision is that even a passive adversary is able to delineate ciphertext boundaries, breaking any passive boundary hiding property. This goes against one of the main security goals of SSH, to hide message sizes. The newest encryption scheme in OpenSSH implements the AE scheme ChaCha20-Poly1305 [Mil15] (not to be confused with the IM scheme using ChaCha20-Poly1305 as its internal AE scheme that is implemented in libInterMAC). 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 lengths by the OpenSSH developers.

5.2 Implementation Details

In this section we highlight the most important implementation choices that were made during the integration of libInterMAC with OpenSSH.

5.2.1 Deviations from RFC 4253

Contrary to OpenSSH's native AES-GCM, Generic Encrypt-then-MAC and ChaCha20-Poly1305 schemes discussed above, 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 serve no 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 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 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.

5.2.2 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 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 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 AE scheme, while the third set specifies the chunk length of the negotiated IM scheme.

5.2.3 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 4.4.

5.3 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 encryption 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 3. 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 4. 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 AEAD schemes and an OpenSSH CBC-mode scheme. 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 3 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 AEAD schemes). The OpenSSH AEAD 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 AEAD 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 4 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 AE scheme, then the best performing IM scheme suffers a 70% performance hit when compared to OpenSSH's native ChaCha20-Poly1305 scheme. The performance difference between IM schemes using AES-GCM as the internal AE scheme and OpenSSH's native AES-GCM scheme is less pronounced, but the IM schemes still suffer from at least a 40% decrease in performance. Furthermore, the OpenSSH 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 Figures 3 and 4 because they perform similarly to the schemes with chunk length 1 greater. In view of the experiments reported in for libInterMAC in Section 4.7, 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.

6 Conclusion

We have 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 easily 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 in the future can be made a fully available encryption option in SSH, and other applications, alongside existing options.

As immediate future work, we plan to carry out further performance testing across a wider range of applications (including interactive terminal sessions) and network conditions. In particular, we wish 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 our IM implementation is indeed constant-time, in the spirit of [ABBD16, ABB+17]. 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 4.6.3).

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 [BT16, LMP17, HTT18]. 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 [RZ18, DF18] that were introduced for analysing more complex secure channel protocols like those employed in SSH.

Acknowledgments

The research of Albrecht was supported by EPSRC grant EP/P009417/1. The research of Hansen was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1). The research of Paterson was supported in part by EPSRC grants EP/M013472/1, EP/K035584/1, EP/P009301/1, and EP/R02314X/1.

References

[ABB+17] 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*, pages 1807–1823. ACM Press, October / November 2017.

[ABBD16] 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, *FSE 2016*, volume 9783 of *LNCS*, pages 163–184. Springer, Heidelberg, March 2016.

[ABF+17] 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.

[ABL+14] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. How to securely release unverified plaintext in authenticated encryption. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 105–125. Springer, Heidelberg, December 2014.

[ADHP16] 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*, pages 1480–1491. ACM Press, October 2016.

[AHP18a] Martin R. Albrecht, Torben Brandt Hansen, and Kenneth G. Paterson. libintermac. https://github.com/himsen/libintermac, 2018. Accessed: 13/02/2019.

[AHP18b] 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: 13/02/2019.

[APW09] 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.

[BDPS12] 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, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 682–699. Springer, Heidelberg, April 2012.

[Ber05] Daniel J. Bernstein. The poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *FSE 2005*, volume 3557 of *LNCS*, pages 32–49. Springer, Heidelberg, February 2005.

[Ber08] Daniel J. Bernstein. ChaCha, a variant of Salsa20. http://cr.yp.to/chacha/chacha-20080128.pdf, 2008. Accessed: 17/11/2018.

[BGR95] Mihir Bellare, Roch Guérin, and Phillip Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 15–28. Springer, Heidelberg, August 1995.

[BKN02] 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*, pages 1–11. ACM Press, November 2002.

[BKR94] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 341–358. Springer, Heidelberg, August 1994.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 159–176. Springer, Heidelberg, October 2012.

[BPS15] Guy Barwell, Daniel Page, and Martijn Stam. Rogue decryption failures: Reconciling AE robustness notions. In Jens Groth, editor, *15th IMA International Conference on Cryptography and Coding*, volume 9496 of *LNCS*, pages 94–111. Springer, Heidelberg, December 2015.

[BR94] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Heidelberg, August 1994.

[BT16] 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, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 247–276. Springer, Heidelberg, August 2016.

[BZD+16] 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.

[Com18] Community. Openssl — cryptography and SSL/TLS toolkit. https://www.openssl.org, 2018. Accessed: 17/11/2018.

[DF18] Jean Paul Degabriele and Marc Fischlin. Simulatable channels: Extended security that is universally composable and easier to prove. *IACR Cryptology ePrint Archive*, 2018:844, 2018.

[DP07] 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.

[DP10] 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*, pages 493–504. ACM Press, October 2010.

[EV16] Guillaume Endignoux and Damian Vizár. Linking online misuse-resistant authenticated encryption and blockwise attack models. *IACR Trans. Symm. Cryptol.*, 2016(2):125–144, 2016. http://tosc.iacr.org/index.php/ToSC/article/view/568.

[Fer05] 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: 17/11/2018.

[FFL12] Ewan Fleischmann, Christian Forler, and Stefan Lucks. McOE: A family of almost foolproof on-line authenticated encryption schemes. In Anne Canteaut, editor, *FSE 2012*, volume 7549 of *LNCS*, pages 196–215. Springer, Heidelberg, March 2012.

[FGMP15] 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, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 545–564. Springer, Heidelberg, August 2015.

[FPR+13] Markus Friedl, Niels Provos, Theo de Raadt, Kevin Staves, Damien Miller, Darren Tucker, Jason McIntyre, Tim Rice, and Ben Lindstrom. OpenSSH 6.2 release notes. https://www.openssh.com/txt/release-6.2, May 2013. Release notes.

[Hou15] R. Housley. Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms. RFC 7696 (Best Current Practice), November 2015.

[HRRV15] 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, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 493–517. Springer, Heidelberg, August 2015.

[HTT18] 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*, pages 1429–1440. ACM Press, October 2018.

[IS09] Kevin Igoe and Jerry Solinas. AES Galois Counter Mode for the Secure Shell Transport Layer Protocol, August 2009.

[Jou06] 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: 17/11/2018.

[LMP17] Atul Luykx, Bart Mennink, and Kenneth G. Paterson. Analyzing multi-key security degradation. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 575–605. Springer, Heidelberg, December 2017.

[LP17] Atul Luykx and Kenneth G. Paterson. Limits on authenticated encryption use in TLS. http://www.isg.rhul.ac.uk/ kp/TLS-AEbounds.pdf, 2017.

[Mil13] 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 weblog. Accessed: 17/11/2018.

[Mil15] Damien Miller. The chacha20-poly1305@openssh.com authenticated encryption cipher draft-josefsson-ssh-chacha20-poly1305-openssh-00, November 2015.

[MV04] David McGrew and John Viega. The Galois/counter mode of operation (GCM). Submission to NIST Modes of Operation Process, 2004.

[NL15] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539 (Informational), May 2015.

[NRS14] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton. Reconsidering generic composition. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 257–274. Springer, Heidelberg, May 2014.

[Pro14] Gordon Procter. A security analysis of the composition of ChaCha20 and Poly1305. *IACR Cryptology ePrint Archive*, 2014:613, 2014.

[Res18] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[RZ18] 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, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2018.

[WMSM11] Andrew M. White, Austin R. Matthews, Kevin Z. Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted VoIP conversations: Hookt on foniks. In *2011 IEEE Symposium on Security and Privacy*, pages 3–18. IEEE Computer Society Press, May 2011.

[YL06] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), January 2006. Updated by RFC 6668.