A Surfeit of SSH Cipher Suites

Martin R. Albrecht
martin.albrecht@rhul.ac.uk

Jean Paul Degabriele
jean.degabriele@rhul.ac.uk

Torben Brandt Hansen
torben.hansen.2015@rhul.ac.uk

Kenneth G. Paterson
kenny.paterson@rhul.ac.uk

ABSTRACT

This work presents a systematic analysis of symmetric encryption modes for SSH that are in use on the Internet, providing deployment statistics, new attacks, and security proofs for widely used modes. We report deployment statistics based on two Internet-wide scans of SSH servers conducted in late 2015 and early 2016. Dropbear and OpenSSH implementations dominate in our scans. From our first scan, we found 130,980 OpenSSH servers that are still vulnerable to the CBC-mode-specific attack of Albrecht *et al.* (IEEE S&P 2009), while we found a further 20,000 OpenSSH servers that are vulnerable to a *new* attack on CBC-mode that bypasses the countermeasures introduced in OpenSSH 5.2 to defeat the attack of Albrecht *et al.* At the same time, 886,449 Dropbear servers in our first scan are vulnerable to a variant of the original CBC-mode attack. On the positive side, we provide formal security analyses for other popular SSH encryption modes, namely ChaCha20-Poly1305, generic Encrypt-then-MAC, and AES-GCM. Our proofs hold for detailed pseudo-code descriptions of these algorithms as implemented in OpenSSH. Our proofs use a corrected and extended version of the "fragmented decryption" security model that was specifically developed for the SSH setting by Boldyreva *et al.* (Eurocrypt 2012). These proofs provide strong confidentiality and integrity guarantees for these alternatives to CBC-mode encryption in SSH. However, we also show that these alternatives do not meet additional, desirable notions of security (boundary-hiding under passive and active attacks, and denial-of-service resistance) that were formalised by Boldyreva *et al.*

1. INTRODUCTION

SSH[1] continues to be an indispensable tool for system administrators. Originally designed as a secure replacement for unencrypted remote login protocols such as Telnet, it has since established itself as the primary protocol for remote login to UNIX environments and has been extended to cover bulk file transfers and other applications. Its growing importance is underlined by Microsoft's announcement in late 2015 to ship OpenSSH in future versions of its Windows operating system.[2]

In 2002, Bellare *et al.* proved a variant of SSH's core component — the Binary Packet Protocol (BPP) — to be secure when it employs CBC-mode encryption [3] (see also [4]). This strengthened our confidence in the then recently standardised protocol. However, in 2009 Albrecht *et al.* [1] presented a surprising yet simple plaintext recovery attack against SSH in CBC-mode. Their attack applied to both SSH as analysed in [3] (where random IVs for CBC-mode were assumed) and as actually specified in RFC 4253 [25] and implemented in OpenSSH, the then-leading implementation of SSH (where chained IVs for CBC-mode are used).

The attack of [1] exploited the fact that in SSH, operations on ciphertexts are not "atomic", but instead an attacker can deliver ciphertexts to be decrypted in a fragmentary fashion. This network-oriented feature of SSH's operation was omitted from the model of [3], and indeed all models of symmetric encryption up to that time. In addition, the attacks relied on the SSH BPP's use of an encrypted length field, rendering the decryption process "plaintext-dependent". The attacks of [1] presented a clear illustration of the significant gap between the theoretical community's treatment of symmetric encryption and its actual usage in building specific secure channels.

The attack in [1] motivated significant follow-up work in two distinct yet closely coupled directions — theoretical modelling and the widespread deployment of improved encryption modes in SSH.

In the first direction, Paterson and Watson [21] analysed SSH's use of CTR-mode, showing that it achieves security in a prototypical model supporting ciphertext fragmentation. The paper [21] inspired the more general and mature treatment of *symmetric encryption supporting fragmented decryption* by Boldyreva *et al.* [10]. That paper introduced general notions formalising confidentiality against fragmentation attacks (which we further extend in the sequel). It also showed that more advanced security notions considered desirable by the designers of SSH, namely boundary hiding security and resistance to certain types of denial-of-service attack, could be achieved at the same time as confidentiality, at low cost using only standard tools.

In the second direction, deployment of improved encryption modes, we have seen a proliferation of alternative modes

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

[2]http://arstechnica.com/information-technology/2015/06/microsoft-bringing-ssh-to-windows-and-powershell/

CCS'16, October 24 - 28, 2016, Vienna, Austria
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4139-4/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978364

of encryption being introduced to SSH. First, the OpenSSH implementation of CBC-mode was quickly patched to prevent the specific attack in [1]. Then many implementations (including OpenSSH and the increasingly popular Dropbear, a lightweight SSH implementation) moved to making CTR-mode the default choice, a selection well-supported by the analysis in [21]. More recently, AES-GCM, generic Encrypt-then-MAC (gEtM henceforth, see Section 2.2) constructions and ChaCha20-Poly1305 have been added to OpenSSH. However, except for CTR-mode, these new modes have not yet been subjected to any serious scrutiny by the research community *in the SSH context*, where, recall, additional attack capabilities beyond those usually assumed in the symmetric encryption setting must be taken into account.

Given the serious nature of the attacks in [1], and the sensitive, high-value nature of at least some traffic protected by SSH, we assert that it is an important problem to study whether the newly introduced modes do lead to a secure channel in SSH. This is also a particularly timely problem to address, in view of ChaCha20-Poly1305 having been promoted to default mode in OpenSSH 6.9 in mid-2015, meaning that it can be expected to quickly become the dominant option.

In this paper, we address this problem and bring the two directions described above — theoretical modelling and deployment of improved modes of encryption — together. We present a systematic analysis of symmetric encryption modes for SSH, provide deployment statistics, new attacks, and security proofs for all widely-used modes. Our main contributions can be summarised as follows:

- After giving an overview of the SSH BPP in Section 2, Section 3 reports deployment statistics based on two Internet-wide scans of SSH servers that we conducted in late 2015 and early 2016. Our scans indicate that many Internet-facing devices now run Dropbear, reducing the previous dominance of OpenSSH in this metric. Overall, Dropbear and OpenSSH implementations dominate in our scans. From our first scan, we found 130,980 OpenSSH and 886,449 Dropbear servers that are still vulnerable to the attack in [1]. Secondly, we observed significant usage of AES-GCM, AES in CTR-mode, and gEtM. We also saw a significant amount of support for ChaCha20-Poly1305. Of these, only CTR-mode currently enjoys a formal security analysis in a security model general enough to capture the attack from [1]. To our surprise, we found 199 distinct combinations of encryption and message authentication code (MAC) algorithms being supported as first preference by at least one server.
- In Section 4, we revisit the original attack from [1]. We demonstrate that the fix implemented in OpenSSH to prevent this attack leaves open a related attack in which byte counting is switched for timing. The detailed attack vector involves timing MAC computations and is reminiscent of the Lucky 13 attack on SSL/TLS [2]. The timing differences are much larger, but the attack depends crucially on an attacker being able to deliver data to the server sufficiently quickly. The attack recovers a limited amount of plaintext from an SSH connection with low probability, but can be amplified over many connections; variants of the attack trade higher success probabilities for a partially known plaintext requirement. The main timing signal exploited in this attack can be eliminated with a simple change to the OpenSSH code base, but a residual timing channel remains and would require more careful patching to completely eliminate it; similarly delicate counter-measures to those for the Lucky 13 attack are needed. Our scans found a further 20,000 OpenSSH servers that are vulnerable to the new attack.
- In Section 5, we provide documentation and formal security analyses for other popular SSH encryption modes, namely ChaCha20-Poly1305, gEtM, and AES-GCM. Our proofs hold for detailed pseudo-code descriptions (included in the full version of the paper) of these modes as implemented in OpenSSH. Extracting these mode descriptions from the single and complex code-path presented by the OpenSSH code-base is itself a contribution, since it renders the modes amenable to formal analysis by us and others. Our proofs use a corrected and extended version of the "fragmented decryption" security model that was specifically developed for the SSH setting by Boldyreva *et al.* [10]. In particular, while Boldyreva *et al.* focused on confidentiality under fragmentation attacks, here we extend their definitions to also cover integrity. Our proofs, then, provide strong confidentiality and integrity guarantees under standard assumptions on the modes' cryptographic components.
- However, in Section 6, we show that these alternative modes do not meet the additional, desirable notions of security (boundary-hiding under passive and active attacks, and DoS resistance) that were formalised by Boldyreva *et al.* [10]. By contrast, it is known from the work of [10], that the complete list of desirable security goals can be met using standard tools at low cost. This leaves an opportunity to further enhance the encryption options available in SSH, a topic that we defer to future work.

Table 1 provides a summary of our results — both positive and negative — for the various encryption modes in SSH.

1.1 Vulnerability Disclosure

We notified the OpenSSH team of our new attack on CBC-mode, applicable to OpenSSH versions 5.2–7.2, on 5/5/2016. We also notified the OpenSSH team of a flaw in MAC processing in the gEtM mode on 22/5/2016. Both issues were addressed in OpensSSH version 7.3, released 1/8/2016.[3]

2. SYMMETRIC ENCRYPTION IN SSH

2.1 The SSH Binary Packet Protocol

The Binary Packet Protocol (BPP) of SSH is defined in Section 6 of RFC 4253 [25]. SSH packets are constructed through a two-step process: payload encoding and cryptographic processing. These are described below.

2.1.1 Encoding.

Encoding proceeds as follows. Firstly, if compression is enabled then the payload (and only the payload) is compressed; Secondly, a length field and a padding length field are prepended to the payload and random padding appended. The length field has 4 bytes and encodes the combined length in bytes of the padding length field, payload and padding. The padding length field has 1 byte and encodes the length in bytes of the padding. The standard mandates that an implementation must be able to support an uncompressed

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

	IND-sfCFA	INT-sfCTF	BH-CPA	BH-sfCFA	n-DOS-sfCFA
CBC	✗	✗	✓	✗	✗
fixed-CBC	✗	✗	✓	✗	✗
CTR	✓	✓	✓	✗	✗
fgEtM	✓	✓	✗	✗	✗
AES-GCM	✓	✓	✗	✗	✗
ChaCha20-Poly1305	✓	✓	✓	✗	✗

Table 1: Security comparison of available encryption modes in SSH.

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

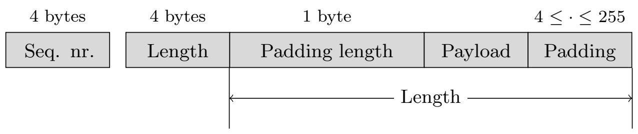

Figure 1: SSH2 packet layout.

2.1.2 *Cryptographic processing.*

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.

The SSH specifications support a variety of encryption algorithms. RFC 4253 [25] defines 7 block ciphers running in CBC mode: 3DES, Blowfish, Twofish, AES, Serpent, IDEA and CAST. In CBC-mode the IV is, for each encryption, set to be the final ciphertext block from the previous encryption; the first IV is chosen uniformly at random. As pointed out in [22, 12] for CBC-mode in general and in [3, 4] for SSH specifically, this potentially makes SSH vulnerable to chosen-plaintext attacks. However, these do not seem to be realisable in practice due to details of the SSH packet encoding. Each cipher may support several different key lengths, e.g. the AES options includes support for key lengths of 128 and 256, denoted by AES128 and AES256, respectively. Of the ciphers defined in RFC 4253, 3DES is required while AES128 is recommended.

In addition to block ciphers, the RC4 stream cipher is supported (and denoted arcfour). RFC 4344 [5] defines CTR-mode options for all the block ciphers mentioned in RFC 4253. Additionally, RFC 5647 [14] defines AES-GCM, i.e. how to use AES-GCM in SSH. In this option, the padding length, payload and padding fields are encrypted and integrity protected. However, the length field is not encrypted, but instead included as Additional Data. Formally, then, AES-GCM deviates from the requirement of RFC 4253 to encrypt the length field.

RFC 4253 specifies as MAC algorithms HMAC-SHA1, HMAC-SHA1-96 (output truncated to 96 bits), HMAC-MD5 and HMAC-MD5-96. Of these, support for HMAC-SHA1 is required and support for HMAC-SHA1-96 is recommended. The later RFC 6668 [9] defines HMAC-SHA2-256 and HMAC-SHA2-512, with the former being recommended and the latter optional. A draft RFC [18] specifies UMAC-32, -64, -96 and -128 for use in SSH. Here, the sequence number is passed as a nonce to UMAC.

2.2 Further Algorithms in OpenSSH

OpenSSH supports additional encryption options beyond those specified in the RFCs. As we shall see in Section 3, these are quite widely adopted and therefore warrant analysis.

Since OpenSSH 6.2, it has been possible to run supported algorithms in an Encrypt-then-MAC mode with the encryption and MAC processing being provided by any of the supported algorithms. Usage is signalled by negotiating an "etm"-MAC during key exchange (if an AE mode is specified, then the special behaviour triggered when negotiating an etm-MAC is disabled). We refer to this mode as gEtM. This option is described briefly in the PROTOCOL file[4] in the OpenSSH codebase but does not seem to be formally documented in the form of an RFC. We have therefore extracted our description of it directly from the OpenSSH source code. The cryptographic processing in the gEtM mode is similar to that of the AES-GCM mode, with the length field not being encrypted but included in the MAC scope; for details, see Section 5.

The RFC draft [17] defines ChaCha20-Poly1305 for SSH. This mode combines ChaCha20, a high-speed stream cipher [7], and Poly1305MAC, a high-speed one-time message authentication code based on a design from [8]. Here, the length field is encrypted using a separate instance of ChaCha20, but the construction otherwise follows RFC 4253. For details, see Section 5. This option has been supported in OpenSSH since version 6.5.

There has been a steady trend towards the elimination of

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

weak algorithms from OpenSSH. OpenSSH 6.6 disabled CBC mode and RC4 by default, while OpenSSH 6.9 promoted ChaCha20-Poly1305 to be the default mode. OpenSSH 7.2 disabled Blowfish-CBC, Cast-CBC and RC4. OpenSSH 4.7 added support for UMAC-64 and OpenSSH 6.2 added support for UMAC-128. OpenSSH 7.2 disabled the MD5-based and truncated-HMAC algorithms.

3. SSH DEPLOYMENT STATISTICS

In this section, we report on SSH deployment statistics, focussing on the mix of different SSH implementations and versions supported by SSH servers with public IP addresses and on their preferred symmetric encryption options.

We ran two IPv4-address-space-wide scans for SSH servers using ZGrab/ZMap [13]. These were conducted from November 11, 2015 to December 1, 2015 and from January 22, 2016 to January 27, 2016. We found about 2^{24} servers in each scan.

In SSH, ciphersuites are negotiated based on the preferences of the client, not the server: the first option from the client's list that is also on the server's list is used. Hence, similarly to recent studies of algorithm deployment in TLS, it is difficult to establish what ciphers and modes are actually used to protect traffic in flight, because only servers and their preferences can be easily queried. Still, the data presented in this section should give a rough estimation of configurations in the wild since the most prominent servers — OpenSSH and Dropbear — share code and default configurations between server and client.

3.1 SSH Implementations and Versions

As Table 2 shows, amongst those servers our scan identified, those self-reporting as `dropbear_2014.66` dominate, with `OpenSSH_5.3` being the second most popular. Overall, the landscape is dominated by OpenSSH and Dropbear servers, with `ROSSSH` being the only server in the top 20 not belonging to either family. Overall, 57.97% (resp. 56.11%) of all servers reported as some version of Dropbear and 37.17% (resp. 39.22%) as some version of OpenSSH in the first (resp. second) scan. In both scans, less than 5% of servers reported as something other than Dropbear or OpenSSH.

software	scan 2015–12		scan 2016–01	
`dropbear_2014.66`	7,229k	(42.0%)	8,334	(47.0%)
`OpenSSH_5.3`	2,108k	(12.3%)	2,133	(12.0%)
`OpenSSH_6.6.1p1`	1,198k	(7.0%)	1,124	(6.3%)
`OpenSSH_6.0p1`	554k	(3.2%)	573	(3.2%)
`OpenSSH_5.9p1`	467k	(2.7%)	500	(2.8%)
`dropbear_2014.63`	422k	(2.5%)	197	(1.1%)
`dropbear_0.51`	403k	(2.3%)	434	(2.5%)
`dropbear_2011.54`	383k	(2.2%)	64	(0.4%)
`ROSSSH`	345k	(2.0%)	333	(1.9%)
`OpenSSH_6.6.1`	338k	(2.0%)	252	(1.4%)

Table 2: Deployment statistics for SSH servers.

3.2 CBC-mode Vulnerabilities

3.2.1 OpenSSH.

The CBC-mode vulnerability reported in [1] was fixed in OpenSSH 5.2. We found 130,980 (resp. 166,572) servers still running older versions of OpenSSH and preferring CBC-mode in the first (resp. second) scan. All these servers are likely to be vulnerable to the attack described in [1]. CTR-mode is now by far the most popular mode offered by OpenSSH servers, being preferred by about 94%.

Considering the attack from Section 4.3 the number of likely vulnerable OpenSSH servers increases to 150,339 (resp. 187,964) in the first (resp. second) scan. These numbers represent the totality of OpenSSH servers preferring CBC-mode.

3.2.2 Dropbear.

Dropbear did not implement a counter-measure to the CBC-mode attack of [1]. Instead `dropbear_0.52` (released 12/11/2008) added support for CTR-mode and made it the default. Hence, any Dropbear server preferring CBC-mode is vulnerable to a variant of the attack from [1] (that is described in the full version).

This includes any version of Dropbear prior to 0.52. We found 886,449 (resp. 816,359) Dropbear servers that prefer CBC-mode in our first (resp. second) scan. Of these all except a few hundred are versions predating 0.52. Overall, then, about 8.4% of Dropbear servers are vulnerable to the CBC-mode attack. The remaining 91.6% prefer CTR-mode. An option to disable CBC-mode ciphers was added in Dropbear 2015.67 (released 28/1/2015).

3.3 Cryptographic Algorithm Diversity

We show the preferred combinations of encryption and MAC algorithms found in our second (January 2016) scan for all OpenSSH, all Dropbear servers, and overall, in Tables 3, 4, and 5. In total, we saw 199 different combinations as first preference at one or more SSH servers, and 155 for OpenSSH. Of course, many of these combinations are used by tiny numbers of servers, but it is still noteworthy that there is so much diversity in deployed algorithms for SSH. We consider a small amount of diversity to be useful, but a large amount to be dangerous, since it brings an increased risk of there being obsolete or insecure options and a higher chance of there being vulnerabilities in the complex code paths needed to support so many options.

We also randomly sampled 2048 IPs which were reporting `dropbear_2014.66` resp. OpenSSH 5.3 to understand what kind of systems were running these services. Based on nmap's OS fingerprinting [15] we speculate that these devices are predominately embedded systems such as routers or firewalls. We did not investigate these systems further. On February 25, 2016 the Shodan.io search engine reports 2,792,391 IPs reporting `dropbear_2014.66`. Almost all of those IPs belonged to an IP block owned by Comcast Cable. However, querying the network block owned by Comcast Cable again on May 2, we only found 83,486 devices listening on port 22.

encryption and mac algorithm	count	
`aes128-ctr + hmac-md5`	3,877k	(57.7%)
`aes128-ctr + hmac-md5-etm@`	2,010k	(29.9%)
`aes128-ctr + umac-64-etm@`	331k	(4.9%)
`aes128-cbc + hmac-md5`	161k	(2.4%)
`chacha20-poly1305@`	115k	(1.7%)

Table 3: Encryption & MAC Algorithms (OpenSSH), `@openssh.com` abbreviated to `@`.

encryption and mac algorithm		count
`aes128-ctr + hmac-sha1-96`	8,724k	(90.4%)
`aes128-cbc + hmac-sha1-96`	478k	(5.0%)
`3des-cbc + hmac-sha1`	321k	(3.3%)
`aes128-ctr + hmac-sha1`	62k	(0.7%)
`aes128-ctr + hmac-sha2-256`	36k	(0.4%)
`aes128-cbc + hmac-sha1`	14k	(0.2%)

Table 4: Encryption & MAC Algorithms (Dropbear).

encryption and mac algorithm		count
`aes128-ctr + hmac-sha1-96`	8,762k	(51.1%)
`aes128-ctr + hmac-md5`	3,887k	(22.7%)
`aes128-ctr + hmac-md5-etm@`	2,012k	(11.7%)
`aes128-cbc + hmac-sha1-96`	570k	(3.3%)
`3des-cbc + hmac-sha1`	413k	(2.4%)
`aes128-ctr + umac-64-etm@`	331k	(1.9%)
`aes192-cbc + hmac-sha1`	258k	(1.5%)
`aes128-ctr + hmac-sha1`	208k	(1.2%)

Table 5: Encryption & MAC Algorithms (overall), `@openssh.com` abbreviated to `@`.

4. ATTACKS ON SSH

We begin by recalling the plaintext recovery attack on CBC-mode in SSH from [1]. In the full version, we give a variant of the attack that applies to Dropbear. We go on to describe the countermeasure to the attack that was introduced in `OpenSSH_5.2`. Finally, we explain how the attack from [1] can be extended to break that countermeasure.

4.1 The Albrecht-Paterson-Watson Attack

We recall the attack from [1] using its text. This attack applied to OpenSSH up to and including version 5.1. Compliant with SSH as described in Section 2.1, OpenSSH 5.1 (and earlier) uses CBC-mode with interpacket chaining and random padding by default. OpenSSH 5.1 decrypts the first block of a BPP packet as soon as it is received and extracts the packet length field from the corresponding plaintext block. OpenSSH 5.1 rejects any packets whose packet length field (measured in bytes) is less than 5 or greater than $256 \times 1024 = 2^{18}$. OpenSSH 5.1 then verifies that the total number of bytes expected in the packet is a multiple of the block size and then continues to accept data on the connection until sufficient data has arrived; here sufficiency is determined by the content of the packet length field and the size of the MAC field. MAC verification then takes place. If the data has been tampered with, this will fail with high probability, leading to a termination of the connection.

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

$$c_i = F_K(c_{i-1} \oplus p_i), \quad i = 1, 2, \ldots, n$$

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

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

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

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

Combining the two preceding equations, we have:

$$p_i^* = c_{i-1}^* \oplus p_1' \oplus IV \qquad (1)$$

From an analysis of the OpenSSH 5.1 source code, it follows that if, after injecting c_i^*, the attacker sees either a termination of the TCP connection over which the SSH connection is running without an SSH error message (indicating a failure of the block length check) or the SSH connection enters a state in which it is waiting for more data, then p_1' must have passed the length check. But the latter only occurs if the packet length field in p_1' lies between 5 and 2^{18}, which in turn occurs only if the first 14 bits of p_1' are all zero.[5] From this information and equation (1), we can calculate the first 14 bits of p_i^*.

To assess the success probability of this attack, we need only calculate the probability that the length check passes. We may assume that c_n, obtained as the last ciphertext block of the preceding BPP packet, acts as a random IV with respect to the block c_i^*. Hence the content of the packet length field in p_1' can be regarded as being a random 32-bit value. Therefore the length check will pass with probability $2^{-14} - 5/2^{18} \approx 2^{-14}$. This attack can be extended to recovering 32 bits of plaintext by counting the number of bytes consumed by the SSH server before it terminates the connection because of a MAC failure. Now the attack's success probability is reduced to 2^{-18} because of the need to pass both the length check and the block length check.

A variant of this attack can be applied to Dropbear. We give the details in the full version of this paper.

4.2 The OpenSSH Countermeasure

Starting with version 5.2, OpenSSH implements a CBC-mode-specific counter-measure against the attack described in Section 4.1, as follows. If the packet length field has the wrong size then a function `ssh_packet_start_discard` is called. The same function is called if the block length check fails and if the MAC check, when eventually performed, fails.

Calling the function `ssh_packet_start_discard` causes the server to wait for a certain number `discard` bytes. Afterwards, once a total of `PACKET_MAX_SIZE` bytes have arrived, `ssh_packet_stop_discard` is called. This function computes a MAC over `PACKET_MAX_SIZE` bytes and then terminates the connection. The overall intention of the countermeasure is to mask the side-channel information used by the attacker in the attack of [1]: now, whenever an error occurs, the implementation waits until `PACKET_MAX_SIZE` bytes have arrived

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

and only then disconnects. This is because of the different settings of the value of `discard` in the different calls to `ssh_packet_start_discard`.

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

4.3 New Attack on OpenSSH CBC Mode

Everything is now in place to describe the new attack. Essentially, it replaces the byte-counting side-channel of [1] with a timing side-channel.

As before, the attacker gathers any target ciphertext block c_i^* from an SSH connection and injects it so that it is interpreted at the server as the first ciphertext block of a new SSH packet. The first 32 bits of the decryption of the target block will be used to construct the packet length field. Then the attacker sends, as quickly as possible, a further `PACKET_MAX_SIZE` bytes to the server.

There are two cases:

1. If either the length check or the block size check on the packet length field fails, then when `ssh_packet_start_discard` is called, it performs a single MAC computation over `PACKET_MAX_SIZE` bytes. For a 16-byte block cipher (AES) this happens with probability roughly $1 - 2^{-18}$ because `PACKET_MAX_SIZE` is set to 2^{18} in the length check and the block size check is a 4-bit condition.
2. If both checks pass, then a MAC verification over the number of bytes indicated by the packet length field is carried out, the MAC verification fails, and then `ssh_packet_start_discard` is called. This involves a second MAC computation over `PACKET_MAX_SIZE` bytes. This case arises with probability roughly 2^{-18}.

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

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

In summary, with overall probability 2^{-19}, the attacker sees a measurable delay for SSH connection termination, indicating that the length and block length checks have passed. This leaks 18 bits of plaintext information to the attacker. These bits correspond to the 14 MSBs and the 4 LSBs in the 32-bit packet length field. This basic form of the attack is already better than random guessing because it provides confirmation of the unknown plaintext bits. Moreover, assuming the target plaintext is sent in an identifiable ciphertext block across many connections, then the attack can be repeated over multiple connections to increase the success probability.

We describe a variant of this attack which recovers more plaintext bits if more precise timing information is available, and variants where the attacker uses partial knowledge of the plaintext to recover further plaintext with higher probability, in the full version of this paper.

4.4 Experimental Results

We verified the conditions of the attack using the current OpenSSH 7.2 and Paramiko[6]. In particular, we verified that under our attack, an OpenSSH server indeed processed 2^{18} resp. $\approx 2^{18} + 2^{17}$ bytes with HMAC if the lengths checks did not pass resp. the MAC check failed. We also performed some basic timing experiments with the following results. If we flipped a bit in the first block of a BPP packet to distort the packet length field, it took about 600 microseconds to compute the MAC on our test system and we waited for 209 microseconds for additional dummy data to arrive on a loopback device (we started timing this in `ssh_packet_start_discard`). The overall time from checking the packet length field to discarding the connection was about 880 microseconds. If we flipped a bit in a later block so that only the MAC check fails, OpenSSH computed two MACs, which took about 1200 microseconds on our target system. The overall time from checking the length field to discarding the packet was about 1500 microseconds. All timings were done on the server and packets from the client were sent over loopback. Hence, these timings reflect a best case scenario for an attacker. Given that the feasibility of timing side-channels over networks is well established for timing signals much smaller than this magnitude [2], we saw no need to pursue further experiments in a more realistic network environment.

4.5 Practical Impact and Countermeasures

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

The simplest mitigation is to stop using CBC-mode encryption in SSH. As our statistics show, other modes that are immune to this style of attack are widely available. Indeed, from the work of [21], we know that CTR-mode is invulnerable to such attacks. In the sequel, we will prove that OpenSSH's ChaCha20-Poly1305, gEtM and AES-GCM modes are not vulnerable either.

[6]http://www.paramiko.org

The attacks can also be mitigated by making the number of bytes passed to HMAC in `ssh_packet_stop_discard` depend on the number of bytes *already* processed by HMAC prior to entering this function, so that the total amount of bytes processed by HMAC always adds up to (roughly) `PACKET_MAX_SIZE`. This would approximately equalise the total time spent on HMAC computations. However, it would not eliminate the timing signal altogether because of the low-level details of HMAC processing. Such small timing differences may remain exploitable, as was the case in the Lucky 13 attack on SSL/TLS [2]. A mitigation of this type was added in OpenSSH version 7.3.

5. SECURITY PROOFS

In this section, we prove security for various symmetric encryption schemes implemented in OpenSSH. We focus on OpenSSH because of its rich variety of schemes and because of its widespread use in practice. We do not consider CTR-mode, because this mode is already covered by [21], but focus on ChaCha20-Poly1305, gEtM and AES-GCM.

5.1 On the Choice of Security Model

Various models exist in the literature for assessing the security of AEAD (Authenticated Encryption with Associated Data) schemes, the most popular being the nonce-based AE security notion that grew out of [6, 23, 24]. However, we will follow the security model put forward in [10], which in turn builds on [3, 21]. Our choice is motivated by there being a substantial gap between a nonce-based AE scheme and a secure channel of the type that SSH aims to construct. While the latter can be built from the former, such a construction is certainly non-trivial, as the latter aspires for significantly more complex functionality and stronger security goals. We explain why in the sequel.

To start with, a secure channel aims to protect against replays and reordering of messages, whereas nonce-based AE security does not. Secondly, protocols like SSH have to operate over TCP/IP which permits the delivery of ciphertexts to the receiver in an arbitrarily fragmented manner. Extending an AE scheme to operate over TCP/IP requires intrusive changes to the decryption algorithm, to the point that even its syntax must become significantly different. It is easy to overlook this aspect and assume that such a transformation is only cosmetic and will not affect security. As a pertinent example, Bellare *et al.* [3] proved (in a very strong sense) the security of a variant of CBC-mode as defined in the SSH BPP, yet it was exactly the mechanism supporting ciphertext fragmentation that allowed the subsequent plaintext-recovery attack by Albrecht *et al.* [1] against both the original CBC-mode construction used in SSH and the variant proven secure by Bellare *et al.*.

Furthermore, support for ciphertext fragmentation can introduce new security vulnerabilities such as exposing ciphertext boundaries and thereby facilitating traffic analysis as well as enabling certain types of denial-of-service attack. Both of these issues were recognised by the SSH designers in the relevant RFC [25] and later incorporated into formal models in [10].

The stronger security guarantees that one obtains from employing these extended security models come at the expense of added complexity in the security analysis. Finally, we note that by considering the full SSH BPP, as opposed to limiting our analysis to the underlying nonce-based AE schemes, we are able to obtain quantitatively better security bounds.

5.2 Notation

Unless otherwise stated, an algorithm may be randomised. An adversary is an algorithm. For any adversary $\mathcal{A}$ and algorithms $\mathcal{X}, \mathcal{Y}, \ldots$ we use $\mathcal{A}^{\mathcal{X}(\cdot), \mathcal{Y}(\cdot), \cdots}$ to denote $\mathcal{A}$'s output after running it with oracle access to algorithms $\mathcal{X}, \mathcal{Y}, \ldots$ and fresh coins. By convention the running time of an adversary refers to the sum of its actual running time and the size of its description. We generically refer to the resources of an adversary as any subset of the following quantities: its running time, the number of queries that it makes to its oracles, and the total length (in bits) of its oracle queries. If $\mathcal{S}$ is a set then $|\mathcal{S}|$ denotes its size, and $y \twoheadleftarrow \mathcal{S}$ denotes the process of selecting an element from $\mathcal{S}$ uniformly at random and assigning it to y.

For any positive integer n, $\{0,1\}^n$ denotes the set of all binary strings of length n, and $\{0,1\}^*$ denotes the set of all binary strings of finite length. The empty string is represented by ε. For any two strings u and v, $|u|$ denotes the length of u in bits, $u \| v$ denotes their concatenation, $u \preceq v$ denotes the prefix predicate which assumes the value true if and only if there exists $w \in \{0,1\}^*$ such that $v = u \| w$, and $u \,\%\, v$ denotes the unique string w such that $u = z \| w$ and z is the longest string simultaneously satisfying $z \preceq u$ and $z \preceq v$. For any string array B, $B[i]$ denotes its i^{th} entry and $B[i \ldots j]$ denotes $B[i] \| \ldots \| B[j]$ with the convention that $B[i \ldots j] = \epsilon$ if $i > j$. Finally, we use $\Pr[\, P : E\,]$ to denote the probability of event E occurring after having executed process P.

5.3 Symmetric Encryption Supporting Ciphertext Fragmentation

5.3.1 *Syntax.*

A symmetric encryption scheme supporting fragmentation $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ with an associated error set $\mathcal{Q}_\perp$ is specified by three algorithms:

- The randomised key generation algorithm $\mathcal{K}$ that returns a secret key K and initial states σ_0 and ϱ_0.
- The stateful encryption algorithm $\mathcal{E}$ takes as input the secret key $K \in \mathcal{K}$, a plaintext $m \in \{0,1\}^*$, and the current encryption state σ, and returns a ciphertext in $\{0,1\}^*$ together with an updated state.
- The deterministic and stateful decryption algorithm $\mathcal{D}$ takes the secret key K, a ciphertext fragment $f \in \{0,1\}^*$, and the current decryption state ϱ to return the corresponding plaintext fragment $m \in (\{0,1,\P\} \cup \mathcal{Q}_\perp)^*$ together with the updated state.

The error set $\mathcal{Q}_\perp$, where $\mathcal{Q}_\perp \cap \{0,1,\P\}^* = \emptyset$, represents the set of possible errors that the decryption algorithm may return to indicate a variety of possible decryption failures. The symbol $\P \notin (\{0,1\} \cup \mathcal{Q}_\perp)^*$ is introduced to denote the end of plaintext messages, thereby enabling an application making use of the decryption algorithm to parse the output uniquely into a sequence of plaintext messages and error messages. For further details the reader is referred to [10].

5.3.2 *Security Definitions.*

A confidentiality notion for symmetric encryption supporting ciphertext fragmentation (IND-sfCFA) was introduced in [10]. In the course of this work, we discovered that the definition of [10] permitted trivial attacks against any scheme. We carefully repair the definition of [10], with a full explanation of the changes required to effect the repair to be found in the full version. We also extend [10], giving an accompanying stateful integrity notion, integrity of ciphertext fragments (INT-sfCTF). We aim for the strongest possible notion in that we require it to be infeasible for an adversary to produce an out-of-sync sequence of ciphertext fragments that returns a plaintext fragment. Note that this may be easier than forging an entire plaintext, and such a notion is implied by ours. Finally, note that unlike the standard setting of nonce-based encryption it does not suffice to prove integrity and chosen-plaintext security (IND-CPA), since without further conditions they do not suffice to guarantee chosen fragment security (IND-sfCFA). In fact such a relation is already invalidated when the decryption algorithm is allowed to return more than one error message [11], as is the case for SSH.

Definition 1 (IND-sfCFA Security). *Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an encryption scheme supporting fragmentation. Let algorithms* LR *and* DEC *be as specified in Fig. 2, where the internal variables are initialised according to algorithm* INI. *For any adversary $\mathcal{A}$ we define its* IND-sfCFA *advantage as:*

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

The scheme $\mathcal{SE}$ is said to be $(\epsilon, \mathtt{R})$-IND-sfCFA secure, if for any adversary $\mathcal{A}$ with resources at most $\mathtt{R}$, its IND-sfCFA advantage $\mathbf{Adv}^{\mathrm{cfa}}_{\mathcal{SE}}(\mathcal{A})$ is bounded by ϵ.

Note that we denote the resources consumed by an adversary by $\mathtt{R}$, without specifying them exactly. As usual, they can be fully specified in terms of various oracle queries, running time of the adversary, etc.

Definition 2 (INT-sfCTF Security). *Let $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be an encryption scheme supporting fragmentation. Let* ENC *and* DEC *be the algorithms specified in Fig. 2, where the internal variables are initialised according to algorithm* INI. *Let* FORGE *denote the event that* DEC *returns a value in $\{0, 1, \P\}^+$, then the* INT-sfCTF *advantage of an adversary $\mathcal{A}$ is given by:*

$$\mathbf{Adv}^{\mathrm{int}}_{\mathcal{SE}}(\mathcal{A}) = \Pr\left[\mathsf{INI}, \mathcal{A}^{\mathsf{ENC}(\cdot),\mathsf{DEC}(\cdot)} : \mathsf{FORGE} \right].$$

The scheme $\mathcal{SE}$ is said to be $(\epsilon, \mathtt{R})$-INT-sfCTF secure, if for any adversary $\mathcal{A}$ with resources at most $\mathtt{R}$, its IND-sfCFA advantage $\mathbf{Adv}^{\mathrm{cfa}}_{\mathcal{SE}}(\mathcal{A})$ is bounded by ϵ.

5.4 Modelling the OpenSSH code

The encryption and decryption processes in OpenSSH are mainly performed in functions `ssh_packet_send2_wrapped` and `ssh_packet_read_poll2`. These present what is essentially a single code-path for the various supported encryption modes: CBC-mode, CTR-mode, ChaCha20-Poly1305, gEtM, and AES-GCM. The code appears to have been developed in a step-by-step fashion as counter-measures to the attack of [1] and extra modes were added. This development approach has arguably resulted in at least one potentially dangerous error being made, concerning when the MAC is checked in gEtM. This is discussed in detail in Section 5.6. One contribution of this paper is to disentangle the various modes in the OpenSSH code and to present them in a clean and self-contained way, thereby rendering them amenable to formal analysis.

In our analysis of OpenSSH's implementation of the SSH BPP symmetric encryption modes, we endeavoured to be as faithful as possible to the OpenSSH code. However, in building our models, we had to make a few simplifications and modifications which we now describe. We assume throughout that both compression and extra padding are disabled.[7] In order to model connection tear-downs we introduce a flag closed as part of the decryption state; once it is set, the decryption algorithm will not return any output. Finally, in our pseudo-code models, we append for every full message returned by the decryption algorithm, a special *end of message* symbol (¶). This is needed in the framework of [10] to demarcate message boundaries, but of course does not exist in the real code.

We next discuss two functions that are common for the cryptographic processing across many encryption options, relating to padding and sanity checking. The padding scheme used is the same for all the options (including stream ciphers) and hence for ChaCha20-Poly1305, AES-GCM and gEtM, despite ChaCha20-Poly1305 and AES-GCM not strictly requiring any padding, and gEtM possibly not needing it, depending on the specific encryption mode negotiated. Padding is required to be random and, in OpenSSH, is computed using the ChaCha20 stream cipher or by other methods depending on configuration. For our purposes, we simply assume the padding to be a uniformly random string of appropriate length. The sanity checks performed on the length field when a BPP packet is received are the same for AES-GCM, gEtM and ChaCha20-Poly1305. This consists of extracting the length from the length field and check whether the length is in the range $[5, 2^{18}]$ and whether the length is a multiple of the block size. For ChaCha20-Poly1305, the packet length cannot be extracted directly from the BPP packet, because the packet length field is encrypted, cf. Section 5.5.

5.5 ChaCha20-Poly1305 in OpenSSH

ChaCha20 is the stream cipher defined in [7], which takes a 32-byte key K, an 8-byte nonce nonce, an 8-byte initial block counter block_ctr, a variable-length plaintext M and a positive integer len and outputs an encryption C of the first len bytes of the plaintext M. Internally, it makes use of a fixed-output-length pseudorandom function, the ChaCha20 block function, a fact which we use in our proofs. We write $\mathsf{C} \leftarrow \mathsf{ChaCha20}(K, \mathsf{nonce}, \mathsf{block\_ctr}, \mathsf{M}, \mathsf{len})$. Reversing the roles of M and C yields the corresponding decryption process. Note that the OpenSSH code combines the nonce and the initial block counter into a single field called the IV, a practice that we think is prone to error. For this reason and better clarity we chose to maintain these two values separate in our code while still accurately reflecting the OpenSSH code.

Poly1305MAC is the one-time MAC defined in [20] which is in turn based on Poly1305-AES from [8]. On input of a 32-byte key K, variable-length string str and a positive integer strlen specifying the size of the string, it returns a 16-byte tag τ. We write $\tau \leftarrow \mathsf{Poly1305MAC}(K, \mathsf{str}, \mathsf{strlen})$.

The generic composition of ChaCha20 and Poly1305MAC is described in RFC 7539 [20] and adapted to SSH in the RFC

[7]An option in OpenSSH allows adding extra padding. This is only used for user password authentication. This option is not modelled in our work.

```
alg. INI
  sync ← true
  $i \leftarrow 0, j \leftarrow 0$
  $C \leftarrow [\,], M \leftarrow [\,]$
  $F \leftarrow \varepsilon, M' \leftarrow \varepsilon$
  $b \twoheadleftarrow \{0,1\}$
  $(K, \sigma, \varrho) \leftarrow \mathcal{K}$
  return
```

```
alg. LR$(b, m_0, m_1)$
  if $|m_0| \neq |m_1|$ return $\varepsilon$
  $(c, \sigma) \leftarrow \mathcal{E}_K(m_b, \sigma)$
  $i \leftarrow i + 1, C[i] \leftarrow c$
  $M[i] \leftarrow m_b \parallel ¶$
  return $c$
```

```
alg. ENC$(m)$
  $(c, \sigma) \leftarrow \mathcal{E}_K(m, \sigma)$
  $i \leftarrow i + 1, C[i] \leftarrow c$
  $M[i] \leftarrow m \parallel ¶$
  return $c$
```

```
alg. DEC$(f)$
  $(m, \varrho) \leftarrow \mathcal{D}_K(f, \varrho)$
  $F \leftarrow F \parallel f, M' \leftarrow M' \parallel m$
  if sync = true
    $j \leftarrow \min(\{n \mid C[1 \ldots n] \npreceq F\} \cup \{i\})$
    if $F \preceq C[1 \ldots j]$
      $m \leftarrow \varepsilon$
    else
      $m \leftarrow M' \,\%\, M[1 \ldots j-1]$
      if $C[1 \ldots j] \preceq F$
        $m \leftarrow M' \,\%\, M[1 \ldots j]$
      if $m \neq \varepsilon$
        sync ← false
  return $m$
```

Figure 2: Algorithms for defining IND-sfCFA and INT-sfCTF security.

draft [17] which defines ChaCha20-Poly1305. In OpenSSH, ChaCha20-Poly1305 is denoted `chachapoly1305@openssh.com` and utilises a 64-byte key and an 8-byte nonce. The nonce consists of a 4-byte sequence number stored as an 8-byte type. The scheme produces a 16-byte MAC tag an encrypts in 64-byte blocks.

Encryption proceeds in three steps: First, the length field is encrypted using ChaCha20, using the first 32 bytes of the key and the initial block counter set to zero. Second, the remaining part of the packet is encrypted using the last 32 bytes of the key and the initial block counter set to one. (That is, two distinct instances of the ChaCha20 algorithm are used to encrypt the two parts.) Third, a MAC tag is computed over the entire encrypted packet. The key used here is obtained from a call to ChaCha20 keyed with the last 32 bytes of the key, the 32-bit sequence number (cast to a 64-bit type) as nonce and an initial block counter value of 0, and an all-zero 32-byte plaintext. Note that the sequence number is not in the MAC scope but is integrity protected implicitly through its role in deriving the MAC key.

Decryption supports ciphertext fragmentation by first decrypting the length field and checking that it satisfies the usual length requirements. Successive ciphertext fragments are then accumulated until the received MAC tag can be verified against the ciphertext. If the MAC tag is valid the remaining portion of the ciphertext is decrypted and the padding removed. A detailed description of ChaCha20-Poly-1305 in pseudo-code is provided in the full version of this paper.

We are now ready to state our theorems regarding the security of ChaCha20-Poly1305 as described above. We provide proofs in the full version of this paper.

Theorem 1 (ChaCha20-Poly1305 is INT-sfCTF secure). *Let ChaCha20-Poly1305 be the scheme described above. Then for any INT-sfCTF adversary $\mathcal{A}_{\text{int}}$ against ChaCha20-Poly-1305, making at most q_e encryption queries totalling μ_e bits, there exists a prf adversary $\mathcal{A}_{\text{prf}}$ against the ChaCha20 block function such that:*

$$\mathbf{Adv}^{\text{int}}_{\textit{ChaCha20-Poly1305}}(\mathcal{A}_{\text{int}}) \leq \mathbf{Adv}^{\text{prf}}_{\text{ChaCha20}}(\mathcal{A}_{\text{prf}}) + \frac{1}{2^{89}}\,, \quad (2)$$

where $\mathcal{A}_{\text{prf}}$ runs in time similar to $\mathcal{A}_{\text{int}}$ and makes at most $(\lceil \frac{\mu_e}{2^9 q_e} \rceil + 1)$ queries.

Theorem 2 (ChaCha20-Poly1305 is IND-sfCFA secure). *Let ChaCha20-Poly1305 be the scheme described in above. Then for any IND-sfCFA adversary $\mathcal{A}_{\text{cfa}}$ against ChaCha20-Poly1305, making at most q_e encryption queries totalling μ_e bits, there exists a prf adversary $\mathcal{A}_{\text{prf}}$ against the ChaCha20 block function such that:*

$$\mathbf{Adv}^{\text{cfa}}_{\textit{ChaCha20-Poly1305}}(\mathcal{A}_{\text{cfa}}) \leq \mathbf{Adv}^{\text{prf}}_{\text{ChaCha20}}(\mathcal{A}_{\text{prf}}) + \frac{1}{2^{89}}\,, \quad (3)$$

where $\mathcal{A}_{\text{prf}}$ runs in time similar to $\mathcal{A}_{\text{cfa}}$ and makes at most $(\lceil \frac{\mu_e}{2^9 q_e} \rceil + 1)$ queries.

It is worth pointing out that the security bounds in the above theorems are independent of the number of decryption queries. These can be contrasted for instance with the bounds obtained in [8], which for the typical parameter values of SSH can become uncomfortably large. Our tighter bounds are a direct consequence of our choice of security definitions, which, albeit offering stronger security, can capture better the extra security that stateful encryption schemes can offer. In Theorems 1 and 2, the Poly1305MAC algorithm can be replaced by any MAC algorithm that is SUF-CMA-secure, with minor modifications to the proofs being required.

5.6 generic-EtM in OpenSSH

The generic Encrypt-then-MAC construction in OpenSSH allows any combination of supported encryption and MAC algorithms to be run in Encrypt-then-MAC mode. However, we note that the actual implementation of gEtM in OpenSSH prior to version 7.3 does not implement Encrypt-then-MAC in the expected way. While the MAC tag of a received ciphertext is computed before decryption commences, it is only compared to the received MAC tag after decryption is complete. Presumably this is as a result of the gEtM mode having been implemented on top of the legacy Encrypt-and-MAC mode in OpenSSH. As a consequence, the decryption function could produce a plaintext-dependent error before the MAC is checked, opening the code up to attacks involving ciphertext manipulation. For example, suppose CBC-mode using PKCS#7 padding were to be at some point added to the roster of available encryption algorithms in OpenSSH. Then the late MAC check would enable a padding oracle style attack to be mounted. We stress, however, that at this point no attack is known exploiting the late MAC check.

It would be possible to prove gEtM secure in our model assuming that any errors thrown by the decryption algorithm can be simulated without knowledge of the secret key (formally, by assuming the existence of an efficient keyless decryption error simulator). But such an approach would neglect the potential vulnerabilities just highlighted. We therefore chose to prove security of a modified gEtM construction, fixed-gEtM (fgEtM) that is identical to gEtM except that it checks the MAC before decryption. We recommended to the developers of OpenSSH that they adopt fgEtM, and they have done so in OpenSSH version 7.3.

The generic Encrypt-then-MAC construction is instantiated with a symmetric encryption scheme and a MAC algorithm, and inherits these schemes' parameters. The construction is formalised as follows.

$\mathcal{SE} = (\overline{\mathcal{K}_c}, \overline{\mathcal{E}}, \overline{\mathcal{D}})$ is any symmetric encryption scheme supported by OpenSSH (not including AES-GCM and ChaCha20-Poly1305). $\overline{\mathcal{K}_c}$ is the (probabilistic) key generation algorithm, which outputs a $\mathsf{keylen_c}$-byte length key K_c, encryption state σ_c and decryption state ϱ_c. We write $(K_c, \sigma_c, \varrho_c) \leftarrow \overline{\mathcal{K}_c}$. $\overline{\mathcal{E}}$ is the encryption algorithm, which on the input of a key K_c, variable-length plaintext M, positive integer ptlen and encryption state σ_c encrypts the first ptlen bytes of M, and outputs a ciphertext C and updated encryption state σ_c. We write $(\mathsf{C}, \sigma_c) \leftarrow \overline{\mathcal{E}}(K_c, \mathsf{M}, \mathsf{ptlen}, \sigma_c)$. $\overline{\mathcal{D}}$ is the decryption function, which on the input of a key K_c, ciphertext C, positive integer clen and decryption state ϱ_c decrypts the first clen bytes of C, and outputs a plaintext M or an error message $\perp$, and updated decryption state ϱ_c. We write $(\mathsf{M}, \varrho_c) \leftarrow \overline{\mathcal{D}}(K_c, \mathsf{C}, \mathsf{clen}, \varrho_c)$. If $\mathcal{SE}$ is stateless σ_c and ϱ_c are both equal to ϵ.

$\mathcal{MA} = (\mathcal{K}_m, \mathsf{MAC}, \mathcal{V})$ is any MAC scheme supported by OpenSSH. $\mathcal{K}_m$ is the (probabilistic) key generation algorithm, which outputs a $\mathsf{keylen_m}$-byte length key K_m. We write $K_m \leftarrow \mathcal{K}_m$. MAC is the tag generation algorithm, which on the input of a key K_m and variable-length string str outputs a ℓ_{tag}-byte length tag τ. We write $\tau \leftarrow \mathsf{MAC}(K_m, \mathsf{str})$. $\mathcal{V}$ is the verification algorithm, which on the input of a variable-length message str and tag τ outputs 1 if and only if $\mathsf{MAC}(K_m, \mathsf{str}) = \tau$, and 0 otherwise. In cases where umac is negotiated, the inputs to the MAC algorithm MAC and verification algorithm $\mathcal{V}$ are augmented with a nonce which is always set to be the sequence number.

Encryption proceeds by first encrypting the padding length field, payload and padding using $\overline{\mathcal{E}}$. Then, a MAC is computed over the resulting ciphertext prepended with the length field and the sequence number. Hence, the OpenSSH implementation of fgEtM does not strictly comply with Section 6.3 of [25], which mandates that the length field be encrypted. In fgEtM, this is because the MAC should be checked before decryption can commence, and the length field is the only indicator of where the MAC tag is located in the stream of ciphertext bytes, so it has to appear in unencrypted form. Decryption is the same process as for ChaCha20-Poly1305 except that we can skip the first step involving decrypting the length field using a separate key. Detailed pseudo-code for fgEtM can be found in the full version of this paper.

We are now ready to state our theorems showing that fgEtM is IND-sfCFA and INT-sfCTF secure. We use the notation $\mathsf{fgEtM}(\mathcal{SE}, \mathcal{MA})$ to indicate that the algorithm pairing $(\mathcal{SE}, \mathcal{MA})$ has been negotiated. The proofs of these theorems can be found in the full version.

THEOREM 3 (fgEtM IS IND-sfCFA SECURE). *Let $\mathcal{SE} = (\overline{\mathcal{K}_c}, \overline{\mathcal{E}}, \overline{\mathcal{D}})$ be a symmetric encryption scheme supported by OpenSSH and let $\mathcal{MA} = (\mathcal{K}_m, \mathsf{MAC}, \mathcal{V})$ be a message authentication code supported by OpenSSH. Then for any* IND-sfCFA *adversary $\mathcal{A}_{\rm cfa}$ against* $\mathsf{fgEtM}(\mathcal{SE}, \mathcal{MA})$*, there exists an* IND-CPA *adversary $\mathcal{A}_{\rm cpa}$ against $\mathcal{SE}$, a* PRF *adversary $\mathcal{A}_{\rm prf}$ against* MAC *and a* SUF-CMA *adversary $\mathcal{A}_{\rm suf}$ against $\mathcal{MA}$ such that*

$$\mathbf{Adv}^{\rm cfa}_{\mathsf{fgEtM}(\mathcal{SE},\mathcal{MA})}(\mathcal{A}_{\rm cfa}) \leq \mathbf{Adv}^{\rm ind\text{-}cpa}_{\mathcal{SE}}(\mathcal{A}_{\rm cpa}) + 2 \cdot \mathbf{Adv}^{\rm sufcma}_{\mathcal{MA}}(\mathcal{A}_{\rm suf}) + 2 \cdot \mathbf{Adv}^{\rm prf}_{\mathsf{MAC}}(\mathcal{A}_{\rm prf}), \quad (4)$$

where adversaries $\mathcal{A}_{\rm cpa}$, $\mathcal{A}_{\rm prf}$ and $\mathcal{A}_{\rm suf}$ use resources similar to $\mathcal{A}_{\rm cfa}$.

THEOREM 4 (fgEtM IS INT-sfCTF SECURE). *Let $\mathcal{SE} = (\overline{\mathcal{K}_c}, \overline{\mathcal{E}}, \overline{\mathcal{D}})$ be a symmetric encryption scheme supported by OpenSSH and let $\mathcal{MA} = (\mathcal{K}_m, \mathcal{T}, \mathcal{V})$ be a message authentication code supported by OpenSSH. Then for any* INT-sfCTF *adversary $\mathcal{A}_{\rm int}$ against* $\mathsf{fgEtM}(\mathcal{SE}, \mathcal{MA})$*, there exists a* SUF-CMA *adversary $\mathcal{A}_{\rm suf}$ against $\mathcal{MA}$ such that*

$$\mathbf{Adv}^{\rm int}_{\mathsf{fgEtM}(\mathcal{SE},\mathcal{MA})}(\mathcal{A}_{\rm int}) \leq \mathbf{Adv}^{\rm sufcma}_{\mathcal{MA}}(\mathcal{A}_{\rm suf}), \quad (5)$$

where adversary $\mathcal{A}_{\rm suf}$ uses resources similar to $\mathcal{A}_{\rm int}$.

5.7 AES-GCM in OpenSSH

The AES-GCM AEAD scheme is specified in [16]. The use of AES-GCM in SSH is described in RFC 5647 [14] and OpenSSH follows this specification. We refer to the use of AES-GCM in SSH by AES-GCM.

For AES-GCM, OpenSSH utilises the AES-GCM encryption and decryption functions from OpenSSL. To accommodate this in our models, we define a nonce-based AEAD scheme $\mathcal{SE}_n = (\mathcal{K}_n, \mathsf{AES\text{-}GCM\text{-}}\mathcal{E}_{\mathsf{openssl}}, \mathsf{AES\text{-}GCM\text{-}}\mathcal{D}_{\mathsf{openssl}})$ that replicates the relevant algorithms of the OpenSSL library. Specifically, the three algorithms will have the following interfaces:

$\mathcal{K}_n$ is a probabilistic key generation algorithm, which outputs a $\mathsf{keylen_n}$-byte key K_n ($\mathsf{keylen_n} \in \{16, 32\}$) and a 12-byte initialisation vector IV. We write $(K_n, \mathsf{IV}) \leftarrow \mathcal{K}_n$. The key generation algorithm is not explicitly used in OpenSSH but we include it here as we need it below.

$\mathsf{AES\text{-}GCM\text{-}}\mathcal{E}_{\mathsf{openssl}}$ takes a key K_n, a 12-byte initialisation vector IV, a variable-length plaintext M, a positive integer ptlen, some variable-length additional data adata and a positive integer $\mathsf{adatalen}$. It encrypts the first ptlen bytes of M, and integrity protects the first ptlen bytes of M and the first $\mathsf{adatalen}$ bytes of adata. $\mathsf{AES\text{-}GCM\text{-}}\mathcal{E}_{\mathsf{openssl}}$ outputs a ciphertext C and an authentication tag τ. We write $(\mathsf{C}, \tau) \leftarrow \mathsf{AES\text{-}GCM\text{-}}\mathcal{E}_{\mathsf{openssl}}(K_n, \mathsf{IV}, \mathsf{M}, \mathsf{ptlen}, \mathsf{adata}, \mathsf{adatalen})$.

$\mathsf{AES\text{-}GCM\text{-}}\mathcal{D}_{\mathsf{openssl}}$ takes a key K, a 12-byte initialisation vector IV, a variable-length ciphertext C, a positive integer clen, some variable-length additional data adata and a positive integer $\mathsf{adatalen}$. It decrypts the first clen bytes of C, and verifies the integrity of the first clen bytes of C and the first $\mathsf{adatalen}$ bytes of adata. $\mathsf{AES\text{-}GCM\text{-}}\mathcal{D}_{\mathsf{openssl}}$ outputs a string M, or one of two error messages: $\perp_{\mathsf{mac}} =$ SSH_ERR_MAC_INVALID or $\perp_{\mathsf{error}}$. The former is returned if the MAC check failed and the latter is returned if the underlying OpenSSL library reports any other error. If an error message is not returned, it means that the MAC tag contained in the ciphertext verified successfully. We write $\mathsf{M} / \perp_{\mathsf{mac}} / \perp_{\mathsf{error}} \leftarrow \mathsf{AES\text{-}GCM\text{-}}\mathcal{D}_{\mathsf{openssl}}(K_n, \mathsf{IV}, \mathsf{C}, \mathsf{clen}, \mathsf{adata}, \mathsf{adatalen})$.

Now we turn to describing the AES-GCM mode in SSH as per the specification in [14] and as implemented in OpenSSH.

During encryption, in calls to $\mathsf{AES\text{-}GCM}\text{-}\mathcal{E}_{\mathsf{openssl}}$, the additional data adata is set to the length field after packet encoding, while the plaintext M is set to the concatenation of the padding length field, the payload and the padding. The initialisation vector IV is set to be a 4-byte fixed field and an 8-byte invocation counter field (ICF), both of which are generated uniformly at random during setup. For each encryption and decryption operation, the ICF is incremented by one while the fixed field is invariant. As with the gEtM construction (and for the same reason), the SSH specification of AES-GCM deviates from Section 6.3 of [25] by not encrypting the length field. Decryption for AES-GCM mode in SSH extracts the length field from the stream of received bytes, sanity checks it, and then calls the $\mathsf{AES\text{-}GCM}\text{-}\mathcal{D}_{\mathsf{openssl}}$ algorithm once sufficient bytes have arrived.

We are now ready to state our theorems regarding the security of AES-GCM. Both theorems rely on the nonce-based AEAD security of the AES-GCM AEAD scheme $\mathcal{SE}_{\mathsf{n}}$ defined previously; for a formal definition of this security notion, see [19]. The proofs of these results are provided in the full version.

Theorem 5 (AES-GCM is IND-sfCFA secure). *For any* IND-sfCFA *adversary* $\mathcal{A}_{\text{cfa}}$ *against* AES-GCM*, there exists an* nAE *adversary* $\mathcal{A}_{\text{nae}}$ *against* $\mathcal{SE}_{\mathsf{n}}$ *such that*

$$\mathbf{Adv}^{\text{cfa}}_{\mathsf{AES\text{-}GCM}}(\mathcal{A}_{\text{cfa}}) \leq 2 \cdot \mathbf{Adv}^{\text{nAE}}_{\mathcal{SE}_{\mathsf{n}}}(\mathcal{A}_{\text{nae}}), \qquad (6)$$

where $\mathcal{A}_{\text{nae}}$ *use resources similar to* $\mathcal{A}_{\text{cfa}}$.

Theorem 6 (AES-GCM is INT-sfCTF secure). *For any* INT-sfCTF *adversary* $\mathcal{A}_{\text{int}}$ *against* AES-GCM*, there exists an* nAE *adversary* $\mathcal{A}_{\text{nae}}$ *against* $\mathcal{SE}_{\mathsf{n}}$ *such that*

$$\mathbf{Adv}^{\text{int}}_{\mathsf{AES\text{-}GCM}}(\mathcal{A}_{\text{int}}) \leq \mathbf{Adv}^{\text{nAE}}_{\mathcal{SE}_{\mathsf{n}}}(\mathcal{A}_{\text{nae}}),$$

where $\mathcal{A}_{\text{nae}}$ *use resources similar to* $\mathcal{A}_{\text{int}}$.

Theorems 5 and 6 still hold true if the nonce-based encryption scheme $\mathcal{SE}_{\mathsf{n}}$ is replaced by any other nonce-based encryption scheme that meets the nAE notion.

6. ADVANCED SECURITY PROPERTIES

As mentioned previously, in addition to confidentiality and integrity, the relevant SSH RFC [25] aims to mitigate against traffic analysis and denial of service. Encrypting the length field in basic SSH modes (such as CBC-mode and CTR-mode) is designed to make traffic analysis based on packet lengths more difficult for passive attackers. However, an active attacker can manipulate ciphertext bits after the length field and observe the number of bytes injected before a MAC error is produced.[8] Denial of service here refers to an attacker flipping bits in the (encrypted) packet length field, causing the receiver to expect a very long ciphertext, leading to a long delay in interaction for the sender and allocation of resources on the receiver's end, cf. [21]. Indeed, the RFC states that implementations "SHOULD check that the packet length is reasonable" and OpenSSH imposes an upper limit of 2^{18} on the 32-bit packet length field.

Boldyreva *et al.* [10] introduced formal security notions to capture these goals, namely BH-CPA, BH-sfCFA, and n-DOS-sfCFA. We discuss these informally here, referring to [10] for the details. Completely hiding plaintext lengths is impossible unless some efficiency is sacrificed. Indeed, simply encrypting the length field does not conceal plaintext lengths, at least not in a sense that is easy to formalise. However, one can hope to hide ciphertext boundaries, meaning that a sequence of ciphertext packets will look like a stream of random bits, which can help to mitigate traffic analysis. Intuitively, the boundary hiding notions of [10] say that, given a concatenation of ciphertexts, an adversary is unable to determine the ciphertext boundaries, and hence can neither determine the number of ciphertexts included in the concatenation nor their individual sizes. Boundary hiding is defined in [10] for passive (BH-CPA) and active (BH-sfCFA) adversaries. In the latter, the adversary additionally has access to an oracle supporting fragmented decryption. On the other hand, n-DOS-sfCFA security as defined in [10] requires that no adversary be able to forge a sequence of ciphertext fragments, totalling n bits, such that the decryption of the sequence returns no output. One can always trivially achieve n-DOS-sfCFA security by imposing an upper limit on the ciphertext size, but this is not ideal as it would necessarily limit the maximum message size. Thus, the technically interesting (and useful) case is when n is significantly smaller than the longest possible ciphertext.

Since both gEtM and AES-GCM expose the length field in the clear it is trivial for an adversary to determine ciphertext boundaries. In contrast, it can easily be shown that ChaCha20-Poly1305 produces ciphertexts that are indistinguishable from random strings (IND\$-CPA) which in turn implies that it is BH-CPA-secure according to a result of [10]. However, ChaCha20-Poly1305 is not BH-sfCFA-secure due to the bit-flipping attack outlined above. As for denial-of-service security, none of the three schemes achieves n-DOS-sfCFA for n smaller than the maximum ciphertext size, since even though the packet length field is integrity protected in all three cases, the MAC tag is only verified after the complete ciphertext (as indicated by the packet length field) has been received. Thus, an adversary could change the contents of the length field to the maximum accepted value (2^{18} for OpenSSH) and the receiver would experience a connection hang until it had received 256 kbytes of ciphertext, at which point the connection would be dropped.

Bringing together the results of this section with those of the previous section justifies the contents of Table 1.

7. CONCLUSIONS

We have provided statistics on the deployment of symmetric encryption options in SSH, and have provided attacks and formal security analysis for the most important options currently implemented in OpenSSH: CBC-mode, ChaCha20-Poly1305, gEtM and AES-GCM (with analysis of the very popular CTR-mode being provided earlier in [21]). Our results are summarised in Table 1. Given the continuing attacks on CBC-mode, our recommendation is that this mode should now be deprecated in SSH.

It is notable that none of the analysed schemes possesses all of the security properties that one might consider desirable for SSH, namely confidentiality and integrity against an adversary with access to a fragmented decryption oracle; boundary hiding against active attacks; and resistance to denial-of-service attacks. Yet such schemes do exist. For example, in [10], Boldyreva *et al.* propose a scheme InterMAC which simultaneously achieves all these properties. Further-

[8]When CBC-mode is used, OpenSSH's counter measure to [1] prevents this simple byte-counting attack, but a timing channel still exists as described in Section 4.3.

more it is easy to show that InterMAC also achieves our new INT-sfCTF notion. InterMAC is relatively efficient both in terms of computation and ciphertext overhead. At heart, it adds an incremental MACing feature to a generic Encrypt-then-MAC construction. In future work, we plan to explore this scheme and possible variants. We will implement it as an additional method in OpenSSH and evaluate its performance in comparison to existing SSH schemes.

Acknowledgments

Degabriele was supported by EPSRC grant EP/M013472/1 (UK Quantum Technology Hub for Quantum Communications Technologies).

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).

Paterson was supported in part by a research programme funded by Huawei Technologies and delivered through the Institute for Cyber Security Innovation at Royal Holloway, University of London, and in part by EPSRC grant EP/M013472/1.

The authors would like to thank Martijn Stam for helpful discussions on security models for the IND-sfCFA setting.

8. REFERENCES

[1] Albrecht, M. R., Paterson, K. G., and Watson, G. J. Plaintext recovery attacks against SSH. In *2009 IEEE Symposium on Security and Privacy* (May 2009), IEEE Computer Society Press, pp. 16–26.

[2] AlFardan, N. J., and Paterson, K. G. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy* (May 2013), IEEE Computer Society Press, pp. 526–540.

[3] Bellare, M., Kohno, T., and Namprempre, C. Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In *ACM CCS 02* (Nov. 2002), V. Atluri, Ed., ACM Press, pp. 1–11.

[4] Bellare, M., Kohno, T., and Namprempre, C. Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Trans. Inf. Syst. Secur. 7*, 2 (2004), 206–241.

[5] Bellare, M., Kohno, T., and Namprempre, C. The Secure Shell (SSH) Transport Layer Encryption Modes. RFC 4344 (Proposed Standard), Jan. 2006.

[6] Bellare, M., and Namprempre, C. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT 2000* (Dec. 2000), T. Okamoto, Ed., vol. 1976 of *LNCS*, Springer, Heidelberg, pp. 531–545.

[7] Bernstein, D. Chacha, a variant of salsa20. http://cr.yp.to/chacha/chacha-20080128.pdf, 2008.

[8] Bernstein, D. J. The poly1305-AES message-authentication code. In *FSE 2005* (Feb. 2005), H. Gilbert and H. Handschuh, Eds., vol. 3557 of *LNCS*, Springer, Heidelberg, pp. 32–49.

[9] Bider, D., and Baushke, M. SHA-2 Data Integrity Verification for the Secure Shell (SSH) Transport Layer Protocol. RFC 6668 (Proposed Standard), July 2012.

[10] Boldyreva, A., Degabriele, J. P., Paterson, K. G., and Stam, M. Security of symmetric encryption in the presence of ciphertext fragmentation. In *EUROCRYPT 2012* (Apr. 2012), D. Pointcheval and T. Johansson, Eds., vol. 7237 of *LNCS*, Springer, Heidelberg, pp. 682–699.

[11] Boldyreva, A., Degabriele, J. P., Paterson, K. G., and Stam, M. On symmetric encryption with distinguishable decryption failures. In *FSE 2013* (Mar. 2014), S. Moriai, Ed., vol. 8424 of *LNCS*, Springer, Heidelberg, pp. 367–390.

[12] Dai, W. SSH2 attack. http://www.weidai.com/ssh2-attack.txt, 2002.

[13] Durumeric, Z., Wustrow, E., and Halderman, J. A. Zmap: Fast internet-wide scanning and its security applications. In *Usenix Security* (2013), vol. 2013.

[14] Igoe, K., and Solinas, J. AES Galois Counter Mode for the Secure Shell Transport Layer Protocol. RFC 5647 (Informational), Aug. 2009.

[15] Lyon, G. F. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning.* Insecure, 2009.

[16] McGrew, D., and Viega, J. The galois/counter mode of operation (gcm). Submission to NIST Modes of Operation Process, 2004.

[17] Miller, D., and Josefsson, S. The chacha20-poly1305@openssh.com authenticated encryption cipher. Working Draft, Nov 2015.

[18] Miller, D., and Valchev, P. The use of umac in the ssh transport layer protocol. https://tools.ietf.org/html/draft-miller-secsh-umac-01, September 2007.

[19] Namprempre, C., Rogaway, P., and Shrimpton, T. Reconsidering generic composition. In *EUROCRYPT 2014* (May 2014), P. Q. Nguyen and E. Oswald, Eds., vol. 8441 of *LNCS*, Springer, Heidelberg, pp. 257–274.

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

[21] Paterson, K. G., and Watson, G. J. Plaintext-dependent decryption: A formal security treatment of SSH-CTR. In *EUROCRYPT 2010* (May 2010), H. Gilbert, Ed., vol. 6110 of *LNCS*, Springer, Heidelberg, pp. 345–361.

[22] Rogaway, P. Problems with proposed ip cryptography. http://www.cs.ucdavis.edu/˜rogaway/papers/draft-rogaway-ipsec-comments-00.txt, April 1995.

[23] Rogaway, P. Nonce-based symmetric encryption. In *FSE 2004* (Feb. 2004), B. K. Roy and W. Meier, Eds., vol. 3017 of *LNCS*, Springer, Heidelberg, pp. 348–359.

[24] Rogaway, P., and Shrimpton, T. A provable-security treatment of the key-wrap problem. In *EUROCRYPT 2006* (May / June 2006), S. Vaudenay, Ed., vol. 4004 of *LNCS*, Springer, Heidelberg, pp. 373–390.

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