Plaintext Recovery Attacks Against SSH

Martin R. Albrecht, Kenneth G. Paterson and Gaven J. Watson
Information Security Group
Royal Holloway, University of London
Egham, Surrey, UK
Email: {m.r.albrecht,kenny.paterson,g.watson}@rhul.ac.uk

Abstract

This paper presents a variety of plaintext-recovering attacks against SSH. We implemented a proof of concept of our attacks against OpenSSH, where we can verifiably recover 14 bits of plaintext from an arbitrary block of ciphertext with probability 2^{-14} and 32 bits of plaintext from an arbitrary block of ciphertext with probability 2^{-18}. These attacks assume the default configuration of a 128-bit block cipher operating in CBC mode. The paper explains why a combination of flaws in the basic design of SSH leads implementations such as OpenSSH to be open to our attacks, why current provable security results for SSH do not cover our attacks, and how the attacks can be prevented in practice.

1. Introduction

Alongside SSL/TLS and IPsec, SSH is one of the most widely used secure protocol suites. SSH was originally designed as a replacement for insecure remote login procedures such as rlogin and telnet. It has since become a general purpose tool for securing Internet traffic. Version 2 of SSH is standardized by the IETF in a series of RFCs [24], [25], [26], [27]. Throughout this paper, we use SSH as shorthand for SSHv2 as defined in these RFCs. Although many different implementations of SSH are available, the OpenSSH implementation [13] dominates, with OpenSSH and its derivatives accounting for more than 80% of SSH implementations on the Internet [20]. We use OpenSSH throughout as shorthand for any version of OpenSSH up to and including version 5.1, the release that was current at the time we carried out this work.

SSH has benefited from a long development process, and the consensus seems to be that it is now a secure design. Moreover, the OpenSSH community claims that OpenSSH has been developed using a rigorous security process [15]. Indeed there are few vulnerabilities that have been discovered in the OpenSSH code over the years [15] and no fundamental design flaws in SSHv2 or OpenSSH have been revealed to date. Only some rather theoretical and easily-circumvented cryptographic attacks have been discovered [9], [1] – see Section 1.1 for more discussion of these. The lack of serious attacks can be taken as evidence for the soundness of the SSH design. Moreover, the SSH Binary Packet Protocol (BPP), the component of SSH that is responsible for providing confidentiality and integrity services to all messages exchanged over an SSH connection, has been subjected to a formal cryptographic analysis using the methods of provable security [1]. The analysis of [1] shows that the SSH BPP as strictly implemented in the way described in the RFCs is theoretically insecure (in the sense that an adversary with sufficient control over the network can distinguish the encryptions of chosen plaintexts). However it also shows that making any one of a number of small modifications to the SSH BPP all result in a protocol that *is* provably secure against chosen ciphertext attacks. Thus the basic design principles behind SSH are supported by the analysis of [1].

In this paper we show that, contrary to this consensus, the SSH BPP specification in [26, Section 6] has serious design flaws that lead directly to plaintext-recovering attacks against SSH. We have implemented a proof of concept of the attacks and variants against OpenSSH. Moreover, we are able to show that the obvious implementation of one variant of the SSH BPP that was proven secure in [1] would also be vulnerable to our attack techniques.

The basic idea behind our attacks is quite simple and can be explained as follows. The SSH BPP most commonly makes use of a block cipher in CBC mode to provide confidentiality[1] and a MAC algorithm on the plaintext data to provide integrity. The protocol format includes a 32-bit packet length field which appears in encrypted form in the first block of ciphertext in an SSH packet. This field is used to determine how much data is expected for a given packet, so this field must be extracted *before* the rest of the packet is received and the MAC can be validated. By sending a target ciphertext block as the first block of a new SSH packet, an attacker can induce an SSH server to treat the resulting plaintext as the first block of a new packet. By then feeding random blocks of ciphertext into the SSH connection in a controlled manner, the attacker can measure how much data

1. For example, RFC 4253 [26] lists `3des-cbc` as being required, `aes128-cbc` as being recommended, a further 12 block cipher variants in CBC mode as being optional, and only one stream cipher, `arcfour`, also optional.

is required before the server considers the whole packet to have arrived, checks the MAC, and produces a MAC failure (with overwhelming probability). This failure is reported as an error message on the SSH connection, and the connection is torn down. The amount of data required to trigger the error message reveals the content of the 32-bit packet length field. Because of properties of CBC mode, this 32-bit value then reveals 32 bits of the target plaintext block (corresponding to the target ciphertext block). An unfortunate feature of this attack (from the adversary's perspective) is the need, on average, to feed about 2^{27} ciphertext blocks into the SSH connection before the MAC check is triggered. But the attack would succeed with probability 1. Notice too that it only requires the attacker to be able to capture a target ciphertext block from the network and then to be able to inject it as the first block of a new SSH packet – no known or chosen plaintexts are needed.

In fact, the actual attacks against the OpenSSH implementation of the SSH RFCs that we develop in Section 3 are a little more complicated than this, and prototyping them involved some difficulties. The main difficulty arises from the fact that the relevant RFC [26] advises that the length field should be sanity checked as soon as the block containing the length field is decrypted, to prevent certain types of Denial of Service (DoS) attack against SSH. Again, the length checks must be applied before the MAC is checked. OpenSSH follows the advice of the RFCs, first checking that the packet length field is at most 2^{18} and then checking that it has a certain divisibility property. The connection is torn down if either check fails. The effect of OpenSSH's length checking is to reduce the success probability of our attack to about 2^{-18}, but also to reduce the maximum number of blocks that the attacker has to inject to only 2^{14}. As we shall see in Section 3, OpenSSH's length checks also allow another attack which can verifiably extract 14 bits of plaintext from a target ciphertext block with probability 2^{-14}. This attack is based on the ability of the attacker to differentiate failures of the two distinct length checks carried out by OpenSSH from one another. Another simple variant of our attacks verifiably recovers 18 bits of plaintext from a target ciphertext block with probability 2^{-18} without requiring the ability to controllably inject random ciphertext blocks.

We report on the experimental validation of our attacks against OpenSSH in Section 4.

As we noted above, our attacks lead to the tear down of the SSH connection, meaning that they cannot directly be iterated to boost the success probability. However, the SSH architectural RFC [24] states that the connection should be re-established in the event of errors. So, if SSH were used to protect a fixed plaintext (e.g. a password) across multiple connections, then the success probability could be increased. A similar scenario was considered for OpenSSL in [7]. Even if our attacks do not lead to reliable recovery of complete plaintext blocks, have low probabilities, and consume large numbers of SSH connections in their iterated forms, they do stand in stark contrast to the intended outcome of using strong cryptography in OpenSSH and to the provable security guarantees promised for variants of SSH in [1].

Fundamentally, our attacks are possible because of a combination of design flaws in the SSH BPP. Firstly, SSH has an encrypted length field in the first block of ciphertext in a packet that is used to determine how much data is expected for a given packet and that must be computed before the MAC can be validated. Secondly, the reliance on CBC mode (even with chained IVs) allows an attacker to inject a target ciphertext block of his choice into a fresh BPP packet as the first block of that packet, and for the decryption of this block in its original position to be related in a known way to its decryption in the fresh packet. These two factors, in combination with the attacker's ability to feed data into an SSH connection blockwise and to detect when a MAC error has arisen, allow an attacker to learn some bits of plaintext corresponding to a ciphertext block of his choice by observing how many blocks are needed to cause a MAC failure. OpenSSH's length checks are merely a complicating factor for the implementation of the attacks that reduce their success probabilities.

Some readers might wonder at this point how we would be able to attack a variant of SSH that was already proven secure in [1]. The basic reason is that, while the authors of [1] recognise that the decryption operation in SSH is not "atomic" and might fail in various ways, their security model does not distinguish between the various modes of failure when reporting errors to the adversary. Moreover, their model does not explicitly take into account the fact that the amount of data needed to complete the decryption operation is itself determined by data that has to be decrypted (the length field). Unfortunately, it seems that real-world cryptographic implementations are more complex than the current security models for SSH handle. We comment on this in greater detail in Sections 5 and 6.

1.1. Related Work

Dai [9] and Bellare *et al.* [1] give attacks against the SSH BPP as defined in [26]. These are effectively distinguishing attacks, that is, attacks that reveal, given a ciphertext, which one of two chosen messages was encrypted by SSH. So these attacks do break SSH by the standards of theoretical cryptography. They require the adversary to know the IV that will be used for encryption in the next SSH packet. In SSH, which uses packet chaining in CBC mode (wherein the last block of ciphertext from the previous packet is used as the IV for the next packet), this may be realistic. The relevant RFC suggests the countermeasure of preceding data-bearing packets with dummy packets – this way, the attacker will not see the required IV until too late (and may

not even be able to tell which block *is* the IV). OpenSSH supports this countermeasure. In contrast with [9], [1], our attacks only require the ability to capture ciphertexts and inject modified ciphertexts into the network (rather than being chosen plaintext), and recover plaintext (rather than being distinguishing attacks). In this sense, our attacks are less demanding for the attacker and have more serious consequences than previous attacks on SSH. The dummy packet countermeasure does not prevent our attacks.

Bellare *et al.* [1] also introduced nomenclature for describing variants of the SSH BPP that is useful for us. *SSH-IPC* (SSH with interpacket chaining) refers to the SSH BPP using CBC mode as defined in [26]. *SSH-NPC* refers to the SSH BPP using CBC mode without packet chaining, using a fresh, random IV for each packet, but with a fixed padding format. Bellare *et al.* [1] present a reaction attack against this variant of SSH, but the relevant RFC recommends using random padding, so the attack should not work against implementations. For example, OpenSSH uses random padding. *SSH-$NPC* refers to the SSH BPP using CBC mode with random per packet IVs and random padding. *SSH-CTRIV-CBC* refers to using CBC mode with IVs generated by encrypting a counter. In this proposal, the IVs are not transmitted, and the encryption and decryption are stateful. *SSH-CTR* refers to the use of counter mode of a block cipher, with the counter being maintained by sender and receiver rather than being transmitted at the start of the packet. Our main attack techniques also apply to both SSH-NPC and SSH-$NPC, even though the latter was proven secure in [1].

Other recent papers conducting analysis of standards and implementations of high-profile secure protocols include [5], [6], [7], [10], [17]. These papers, like ours, highlight the problems that arise in protocol specifications with respect to implementation details having the potential to undermine security. They also show that, in order to evaluate security, it is not enough to look at the specification alone – rather, one must look at how the specifications have been implemented in order to gauge whether an attack idea will work against a real system. These papers, also like ours, make use of what might be termed software-based side channels in order to mount their attacks. For example, [7] used timing differences in processing padding and MAC failures to attack OpenSSL, while [10], [17] exploited ICMP error messages of various kinds to attack encryption-only configurations of IPsec.

A timing side-channel analysis of SSH [21] showed that SSH can still leak significant information about users' passwords when it is used to protect an interactive session. Our attacks apply irrespective of what type of session is being protected.

Several papers discuss security of symmetric encryption schemes against blockwise adaptive attackers, beginning with [12]. However, none of these papers consider how errors arising during decryption can undermine security, so none of the work to date in this line of research can be used directly to model the security of SSH in a way that is sufficiently complete to capture our attacks. Moreover, some of our attacks do not require blockwise control but only the ability to inject a single block of ciphertext at an appropriate point in an SSH connection.

1.2. Paper Organisation

Section 2 provides an overview of the SSH BPP protocol and the OpenSSH implementation of this protocol, focussing on how decryption is performed in OpenSSH. Section 3 presents our new attacks on OpenSSH and Section 4 describes our proof-of-concept implementation of the attacks. Section 5 extends our attacks to cover some of the variants of SSH that were proposed in [1]. Section 6 provides recommendations on how to prevent our attacks. These range from selecting modes of operation other than CBC mode through to a more radical overhaul of the SSH BPP. Section 7 presents our final conclusions.

2. The SSH Binary Packet Protocol

The Binary Packet Protocol (BPP) of SSH is defined in [26, Section 6]. Informally, it uses a "encrypt and MAC" construction, wherein the plaintext is both encrypted (to produce the ciphertext) and integrity protected (by using a MAC algorithm). The MAC value is appended to the ciphertext.

In more detail, a payload message is first *encoded* by prepending a packet length field and padding length field and appending some padding. The packet length field is 4 bytes in length and contains the total length (in bytes) of the encoded packet excluding the packet length field itself. The padding length field is 1 byte in length and contains the total number of padding bytes. A minimum of 4 padding bytes must be added, the padding should be random, and the padding must ensure that the encoded data ends on a block boundary. The maximum length of padding is 255 bytes; variable length padding may help frustrate traffic analysis [26].

This encoded message is then encrypted. The final ciphertext is the concatenation of the encoded-then-encrypted message and a MAC value, with the MAC value being computed over the concatenation of a 32-bit packet sequence number and the encoded (but not encrypted) message. The sequence number is set to zero at the start of an SSH connection, and is incremented after each packet. It is not sent over the channel but is maintained separately by both communicating parties. Figure 1 shows the BPP packet format schematically. Notice that the length field is encrypted, with the rationale that this makes it harder for an attacker to detect BPP packet boundaries and so perform traffic analysis.

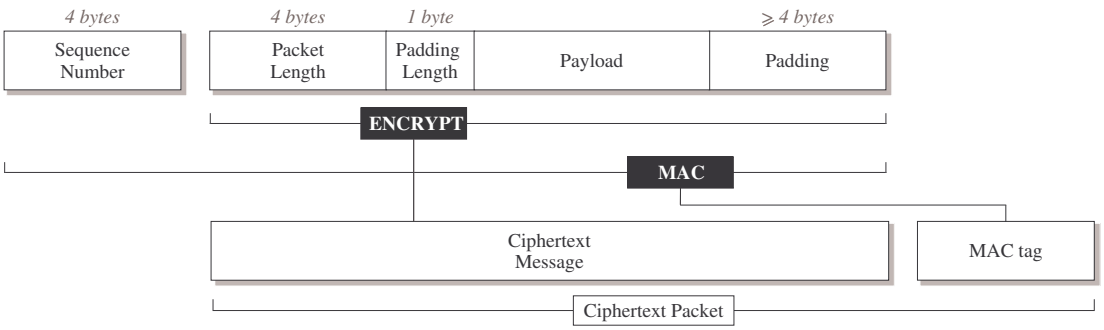

Figure 1. SSH BPP packet format and cryptographic processing

As we mentioned in the introduction, the SSH RFC [26] mandates support for `3des-cbc`, recommends support for `aes128-cbc`, and lists a further 12 block cipher variants in CBC mode as being optional. Only one optional stream cipher is listed, `arcfour`. The RFC mandates the use of initial packet chaining with CBC mode, so that the last block of ciphertext from packet $i-1$ on a connection is used as the IV for CBC mode for packet i on the connection. In this way, the packets on a connection form a single data stream. Notice that the length field will be contained (in encrypted form) in the first block of each packet.

Next we consider how decryption takes place in the SSH BPP. Since there is no length indicator for a BPP packet other than the content of the packet length field, any SSH implementation must decrypt the first ciphertext block to obtain that field and use it to determine how much data to accept before deciding that a complete BPP packet has arrived and moving on to perform the MAC check. Thus we may expect that an SSH implementation will await further data, unless sufficient data has already arrived to complete the packet. In general then, an attacker may be able to delay the MAC check for a particular BPP packet by delaying the data on the SSH connection.

As we mentioned in the introduction, the SSH RFCs recommend sanity checking of the length field. The following quotes are from the beginning of Section 6 and Section 6.1 of [26]:

> *"Note that the 'packet_length' field is also encrypted, and processing it requires special care when sending or receiving packets."*
>
> *"The minimum size of a packet is 16 (or the cipher block size, whichever is larger) bytes (plus 'mac'). Implementations SHOULD decrypt the length after receiving the first 8 (or cipher block size, whichever is larger) bytes of a packet."*
>
> *"... implementations SHOULD check that the packet length is reasonable in order for the implementation to avoid denial of service and/or buffer overflow attacks."*

Exactly when this last check is to be performed is not made explicit in [26], but the natural interpretation is to do the check as soon as the first block of plaintext has been decrypted. Otherwise, denial of service attacks based on manipulating the packet length field would not be prevented. Since the MAC calculation is done on the entire plaintext message (prepended with the 32-bit sequence number), it is clear that, if this interpretation of the RFC is made, then any length checks will be done before the MAC check. Nor is the term "reasonable" defined in the RFCs. However, [26] recommends that

> *"all implementations MUST be able to process packets with an uncompressed payload length of 32768 bytes or less and a total packet size of 35000 bytes or less.... Implementations SHOULD support longer packets, where they might be needed."*

So a reasonable interpretation of the RFC might be to check that the packet length is at most somewhere in the region of 35000 bytes.

Error handling for the BPP is specified in [26, Section 11]. An SSH connection should terminate whenever a transmission error occurs or MAC verification fails, but the terminating entity may send an informative message to its peer. Section 9.3.5 of the SSH architectural RFC [24] states that the connection should be re-established in the event of "transmission errors or message manipulation". Presumably, the latter is to be detected via the MAC check.

2.1. The OpenSSH Implementation of the BPP

OpenSSH follows [26] fairly closely in its implementation of the BPP. It uses CBC mode with interpacket chaining and random padding by default. It maintains sequence numbers, and MAC-protects the correct fields. OpenSSH decrypts the first block of a BPP packet as soon as it is received. OpenSSH then performs the following checks, in the order described.

2.1.1. Length Check. The following excerpt of code from the file `packet.c` shows OpenSSH's implementation of a length check:

```
if (packet_length < 1 + 4 ||
    packet_length > 256 * 1024) {
    buffer_dump(&incoming_packet);
    packet_disconnect("Bad packet length
                %d.",packet_length); }
```

The `packet_disconnect` function terminates the session and sends an `SSH2_MSG_DISCONNECT` SSH error message over the connection. This message contains the passed string. Thus we see that the OpenSSH implementation does not accept any packets whose packet length field is less than 5 or greater than $256 \times 1024 = 2^{18}$. The value 2^{18} is presumably chosen to limit the effectiveness of DoS attacks; the value of 5 is the minimum possible value of this field, given the mandatory presence of a padding length byte and 4 bytes of padding.

This check is consistent with the recommendations of [26] that a total packet size of 35000 must be supported, and that longer packets should be supported.

2.1.2. Block Length Check. OpenSSH then verifies that the total number of bytes expected in the packet is a multiple of the block size:

```
if (need % block_size != 0)
        fatal("padding error:
        need %d block %d mod %d",
        need, block_size, need % block_size);
```

Here,

```
need = 4 + packet_length - block_size
```

denotes the number of bytes still awaited in this packet and can be calculated from the packet length field. The `fatal` function terminates the session but no error messages are sent on the connection (in contrast to the previous check). Instead, the passed string is only logged locally. However, this error will result in the termination of the TCP connection over which SSH is running.

2.1.3. MAC Check. OpenSSH then continues to accept data on the connection until sufficient data has arrived. MAC verification then takes place. The following OpenSSH code controls how much data needs to be received before the MAC check will be done:

```
if (buffer_len(&input) < need + maclen)
        return SSH_MSG_NONE;
```

Here `buffer_len(&input)` measures in bytes the amount of data received for the current packet and `maclen` denotes the length in bytes of the SSH MAC field. If the subsequent MAC check fails, then the `packet_disconnect` function is called with the particular error message "`Corrupted MAC on input.`" being sent on the connection. If this check passes, then OpenSSH performs a series of further checks. These need not concern us here.

Summarising the above, we can see that OpenSSH's decryption operation involves a number of distinct steps, each step potentially resulting in a different type of error and subsequent behaviour.

3. Attacking OpenSSH

3.1. Notation

Before describing our attacks against OpenSSH let us first define some notation. 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 the SSH BPP 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

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

3.2. Recovering 14 Plaintext Bits

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. We have:

$$p_i^* = c_{i-1}^* \oplus F_K^{-1}(c_i^*).$$

Consider now an attacker who simply injects the single block c_i^* as the first block of a new packet on the SSH connection[2]. Let c_n denote the last ciphertext block of the preceding packet on the connection. This block will be used as the IV for the new packet, and hence OpenSSH will compute as the first block of plaintext for this new packet:

$$p_1' = c_n \oplus F_K^{-1}(c_i^*).$$

Combining the two preceding equations, we have:

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

2. Note that since the attacker cannot necessarily detect where one BPP packet ends and the next begins, there is a chance that this injected block is not processed as the first block of a new packet. This is not usually an issue in practice because the attacker can wait until the SSH connection is quiet before beginning his attack.

If, after injecting c_i^*, we see 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 we know that p_1' must have passed the length check[3]. 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. 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 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 $2^{-14} - 5/2^{18} \approx 2^{-14}$.

Note that the attacker can verify when he has been successful in his attack, so this attack has the same success probability but is more powerful than the attack that simply guesses 14 bits of plaintext without being able to verify whether the guess is correct.

3.3. Recovering 32 Plaintext Bits

With exactly the same attack as above, if the SSH connection enters a wait state, then we can deduce that both the length check and the block length check have passed. When $L = 16$ (e.g. with `aes`), this implies that the first 14 bits of the length field in p_1' will all be zero, and that the last 4 bits of this field encode the value 12. In turn, this yields 18 bits of p_i^* using equation (1). Reasoning as before, this event will arise with probability roughly 2^{-18}. (When $L = 8$, the last 3 bits of the length field should encode the value 4, the event has probability roughly 2^{-17}, and reveals 17 bits of p_i^*.) We next explain how the attacker can continue the attack to extract more plaintext.

Recall that, if the length check and block length check pass, then the SSH connection will continue to wait for more data until the following condition is no longer satisfied:

```
if (buffer_len(&input) < need + maclen)
        return SSH_MSG_NONE;
```

Once this test fails, the MAC check will be triggered. So the attacker continues his attack by injecting `maclen` random bytes followed by a sequence of random L-byte blocks into the SSH connection, waiting after the injection of each block to see if the SSH connection is terminated because of a MAC error. This termination occurs after at most $2^{18}/L$ blocks are injected, and when it occurs, the amount of data fed into the connection up to this point reveals the value of `need`, and hence the exact value of the 32-bit packet length field in p_1' using the formula:

```
need = 4 + packet_length - block_size.
```

Knowing this 32-bit value, the first 32 bits of p_i^* can be recovered using equation (1). The overall success probability for the attack is 2^{-18} when $L = 16$ (and 2^{-17} when $L = 8$).

In essence, this attack exploits the encrypted length field and the wait state that arises in OpenSSH provided the length checks pass. We rely on the MAC error to reveal the amount of data expected in the packet, and, through this, the content of the length field. Because of the use of CBC mode, this leaks information about the target ciphertext block. This kind of attack seems endemic in any protocol that combines the following features: the use of an encrypted length field; CBC mode; a reliable transport allowing the attacker to deliver small amounts of data at a time; and signalling of errors. We comment on this in more detail in Section 6.

3.4. Iterating the Attack

Both the attacks above result in the SSH connection being terminated with high probability at each attempt. Suppose, however, that OpenSSH is used to protect plaintexts that contain some fixed bits in fixed, known positions across multiple connections. Suppose further that some of these positions coincide with the first 32 bits in a block. For example, this may be the case if OpenSSH is used to protect a user password for a remote login. Then the attacks above can simply be iterated in order to increase the success probability of extracting the fixed plaintext bits in selected positions. A similar attack to this was considered for OpenSSL in [7]. This variant would be particularly serious for SSH clients that automatically perform connection re-establishment in the event of errors, as recommended in [24].

As described, this iterated version would consume on average 2^{18} SSH connections in order to recover 32 plaintext bits. By more carefully selecting the point at which each target ciphertext block is injected, an attacker can reduce the number of connections used to at most $2^{14} + 2^4$ (in the case of a 128-bit block cipher). The attack is now split into three phases. In the first phase, the attacker recovers the first 14 bits of the 32 target bits as follows. He observes each new SSH connection and waits until an IV appears on the channel which guarantees that a different value of the first 14 bits of the length field will be set (in comparison to previous connections) if the current value of the target ciphertext block c_i^* were to be injected as the first block of the next SSH packet. This requires the attacker to maintain a size 2^{14} table of bit values, each entry indicating whether a particular value consisting of the first 14 bits of $c_{i-1}^* \oplus IV$ has been used on

3. The SSH connection cannot terminate with a MAC failure at this point: each BPP packet contains at least one ciphertext block plus a MAC field, so the connection will not yet have received sufficient data to reach the stage of performing a MAC check.

a previous connection or not. After at most 2^{14} connections, the length check will pass, and the attacker can recover the first 14 bits of plaintext (as in Section 3.2). In the second phase, the attacker can exploit his knowledge of these 14 bits: he now observes each new SSH connection and waits until an IV appears on the channel which guarantees that the length check will be passed if the current target ciphertext block c_i^* is injected. This just involves comparing the first 14 bits of $c_{i-1}^* \oplus IV$ with the 14 plaintext bits recovered so far. By also working with bits 28-31 of $c_{i-1}^* \oplus IV$, the attacker can make sure that a different value of the least significant 4 bits of the length field will be set when c_i^* is injected for each connection in the second phase. On average, the attacker will have to observe about 2^{18} SSH packets until a suitable IV appears for each second phase connection. Then after at most 2^4 second phase connections, both the length check and the block length check will pass, at which point the current connection will enter its wait state. The third phase is the same as the last part of our 32-bit-recovering attack.

Notice that if the attacker has partial knowledge about plaintext – for example that it must be printable ASCII – then the success probabilities for all of our attacks can be increased using similar tricks. We are grateful to an anonymous referee for pointing this out. Note too that the use of data compression on the SSH connection may reduce the utility of the plaintext bits recovered in our attacks, and will interfere with the connection-conserving variant of our iterated attack.

The attacker does not need to be able to control the IV used for the attack packet in any of the attacks described above. Instead, he merely needs to be able to learn the value of that IV at an appropriate point in the attack. Thus the measure recommended in [26] for preventing Dai's attack, in which the transmission of the relevant IV is simply delayed, does not prevent our attacks.

4. Experimental Validation

We implemented a proof-of-concept of our attacks against OpenSSH using Scapy [18] and tested it in a virtual TCP/IP network. Using a local virtual network has the advantage that we can ignore any latency issues, since the transport is almost instantaneous. There is no reason to think that our results would not be applicable to real networks.

For our experiments we used OpenSSH server version 4.7 as shipped with Debian GNU/Linux Lenny (the attack will also apply to more recent versions of OpenSSH up to and including OpenSSH 5.1). However, we made one patch to the server in order to increase the success probability of the experiments: we modified the packet length field so that its 12 most significant bits were set to zero before the length check was performed. The only impact of this change is that the probability of passing this check is increased from 2^{-14} to roughly 2^{-2}. Making this patch allowed us to more easily test that the subsequent parts of our attacks were working as anticipated, and did not influence the behaviour of the server in our attack in any other way. Without this modification, testing and development would have been far more time-consuming. Of course, a real attacker does not get to increase his success probability in this way! We later removed this modification.

We first open up a legitimate connection with the server, negotiating `aes-cbc` and `hmac-md5` (having a 16-byte MAC value). Then, once the SSH connection is established, and in a separate thread, we inject the target ciphertext block into the SSH connection (and then subsequent bytes and blocks should the length check and block length check pass). This thread also monitors the server for responses.

The various decryption checks and states of OpenSSH showed up on the network as follows:

Length Check (c.f. Section 2.1.1): since the server sends a `SSH2_MSG_DISCONNECT` message including a particular error message, the failure of this check is indicated by the size of the reply packets on the SSH connection.

Block Length Check (c.f. Section 2.1.2): since the SSH server terminates the connection without any error message in this case, the failure of this check is indicated by the presence on the network of a TCP FIN packet without any payload.

MAC Check (c.f. Section 2.1.3): since the server sends a `SSH2_MSG_DISCONNECT` message including a particular error message, the failure of this check is indicated by the size of the reply packets on the SSH connection. We may distinguish this failure from the length check failure above because the SSH connection will have passed through an intermediate wait state in our attacks if we get as far as the MAC check.

For our attacks to work, we rely on the behaviour of two different services: the TCP/IP stack of the operating system and the SSH server. The TCP/IP stack may or may not acknowledge any packet injected into the TCP connection with a TCP ACK before the SSH server has the opportunity to react to it. Consequently, observing a TCP ACK on the wire is not a sufficient indication that both the length check and block length check have passed and that the desired wait state has been reached. Instead, we must detect this state by the *absence* of either a TCP FIN packet without payload or an SSH disconnect message. This test is therefore somewhat sensitive to the latency of the network, since we need to wait for a timeout. This only affects the time taken to carry out the attacks.

For our first attack, if a TCP FIN packet without payload is observed then we know that the length check has passed but the block length check has failed. If no TCP FIN packet without payload or SSH disconnect message is observed within a reasonable time-out period, then we know that both

the length check and the block length check have passed. In either case, at least 14 bits of plaintext can be recovered.

Of course, if no TCP FIN (with or without an SSH disconnect message as payload) is observed, we may assume that the wait state has been reached, at which point 18 bits of plaintext (for $L = 16$) can be recovered and our second attack can begin. We then feed single blocks of ciphertext to the SSH server until a `SSH2_MSG_DISCONNECT` message is observed. This indicates a MAC failure after enough data was received. At this point, 32 bits of plaintext can be recovered.

Because the final MAC check could be on a potentially large message (2^{18} bytes), we must take care not to feed blocks to the SSH server too quickly during the second attack. Otherwise, the SSH server may have stopped accepting data and be engaged in MAC processing when we think it is still waiting for further data. This would lead us to overestimate the amount of data that needs to be sent to the server before the MAC check is triggered, distorting the low order bits of our calculation of the content of the packet length field. Thus we throttled the speed of our data injection. This only affects the amount of time needed to carry out the attack.

Using this proof-of-concept code (including the server patch to increase the success probability), we were able to reliably recover the value of the packet length field after decryption, and hence recover the first 32 bits of the original plaintext block corresponding to the target ciphertext block. When we removed the server patch, we still observed several successful 14-bit recovering attacks, but did not observe any successful probability 2^{-18} attacks in the experimental time available.

One of the main challenges for building an exploit based on our proof-of-concept code would be to find a service which tolerates SSH connection failures and reconnects on these failures. One such client is for instance the Fuse SSHFS [19] implementation which accepts a `reconnect` flag and doesn't seem to care about the nature of the connection termination. Even then, the attacks require large numbers of reconnects and may consume a lot of bandwidth.

5. Comparison with Proven Security of SSH

Bellare *et al.* [1] performed a formal security analysis of the SSH BPP. Their work was inspired by the distinguishing attack of Dai [9] against SSH-IPC (SSH with interpacket chaining) and their main focus was to find a secure alternative to SSH-IPC that necessitated minimal changes to the BPP.

The security analysis of [1] considers a general Encode-then-Encrypt-and-MAC scheme. The decryption algorithm of such a scheme takes a complete ciphertext as input and proceeds by first decrypting, then decoding and finally checking the MAC. Whilst recognising that different failure conditions might occur during decryption, the formal decryption algorithm for such a scheme in [1] only produces one possible error message ("$\perp$"). Therefore the model in [1] is intended to not allow the adversary to distinguish between the different types of error. The model does not allow for the possibility that the amount of data needed to complete the decryption process might be governed by data that is produced during the decryption process itself (namely the packet length field). Indeed, the packet length field does not appear at all in the model used in [1]. The analysis in [1] models a connection tear-down in the event of an error, by having decryption always output "$\perp$" in response to any decryption query made after the first error has arisen.

However, as we have seen, in any conceivable implementation of SSH, the decryption process *must* depend on the initially encrypted packet length field. Moreover, in OpenSSH, the adversary *is* able to distinguish between the different error types. Finally, in practice, an attacker is able to feed ciphertext blocks one by one to the decryption process, a feature exploited in our 32-bit-recovering attack. These differences between the theoretical model and the implementation reality explain why some of the schemes proven secure in [1] would be insecure in practice if implemented by extending OpenSSH in the natural way.

Even if the model of [1] only allows one type of error message, it unintentionally introduces a timing channel which can be used to distinguish the different types of error arising during decryption. This channel arises because each of the stages of decrypting, then decoding, and finally checking the MAC is allowed to individually output the single error message in the model of decryption used in [1]. But these stages necessarily follow one after another in series and would take different amounts of time to complete in any implementation. So even having a single error message may not be enough in practice to disguise the reasons for failure, and would still leave open the possibility of attack.

In the remainder of this section, we estimate the extent to which the various SSH BPP enhancements discussed in [1] would be vulnerable to our attacks if they were implemented in OpenSSH in the most obvious and natural way.

5.1. SSH-NPC and SSH-$NPC

SSH-NPC and SSH-$NPC both use CBC mode with random per packet IVs (No Packet Chaining). This incurs a penalty in that the IV now needs to be sent as an additional ciphertext block for each packet. SSH-NPC uses fixed padding. Bellare *et al.* showed that the use of fixed padding allows a "reaction" attack against the scheme that leaks a small amount of information about the relationship between two plaintexts to the attacker. To prevent this attack Bellare *et al.* introduced SSH-$NPC, proving it secure. This BPP variant uses random padding, as recommended by the relevant RFC [26], in combination with random IVs.

Suppose that SSH-NPC and SSH-$NPC were implemented in such a way that the attacker can distinguish between a failure of an OpenSSH-style packet length check and a MAC failure. This would be the case if OpenSSH was extended in the obvious and natural way to support IVs transmitted on the wire. Then our first two attacks from Section 3 would be applicable directly to both SSH-NPC and SSH-$NPC, with the attacker simply making a random choice of IV and injecting this along with the target ciphertext block, instead of relying on the IV being determined by the last block of the previous packet. The attacks' success probabilities would be as in Section 3. In our third attack (where the attacker attempts to learn plaintext bits that are fixed across multiple SSH connections), the attacker's ability to control the IV would allow him to systematically explore the space of IVs to obtain a deterministic attack requiring at most $2^{14} + 2^4$ connections and one injected ciphertext block per connection (for all but the final connection) to recover 32 bits of plaintext.

An attacker can also exploit his control over the IV in SSH-NPC and SSH-$NPC to obtain a simple distinguishing attack that has a success probability of 1. We sketch this attack here. The attacker chooses two BPP plaintexts $M_0 = p_1, p_2$ and $M_1 = p_1, p_2'$ such that the first 32 bits of p_2 are all zero while the first 32 bits of p_2' are the binary representation of the value 32. Here, we assume that block p_1 contains an appropriate length field and padding length field, and blocks p_2, p_2' contain appropriate SSH padding. We assume now that the attacker observes the BPP packet $c_0, c_1, c_2, \mathit{MAC}$ corresponding to either M_0 or M_1 on the SSH connection. The attacker then simply injects blocks c_1, c_2 as the first two blocks of a new BPP packet into the SSH connection, followed by a random MAC value. Here c_1 is interpreted as the IV and the decryption of c_2 as the first block of the plaintext packet. From our choice of p_2, p_2' it is clear that if M_0 was encrypted under C, then the injected blocks will cause a failure of the packet length check and termination of the SSH connection, while if M_1 was encrypted under C, then the SSH connection enters into its wait state. Thus the adversary can distinguish which message was encrypted.

This attack still works even if the length check and the block length check used by OpenSSH are combined into a single uniform check.

Random IVs are therefore not the solution to securing the SSH BPP.

5.2. Further Provably Secure Variants

Bellare *et al.* also proposed and proved secure three further variants of the SSH BPP: SSH-CTR, SSH-CTRIV-CBC and SSH-EIV-CBC. SSH-CTR uses counter mode but padding is still used in order to minimise changes to the BPP. SSH-CTRIV-CBC and SSH-EIV-CBC use CBC mode with an IV that is either the encryption of a counter or the encipherment of the last ciphertext block, respectively.

These three schemes are all resistant to our plaintext recovery attacks. SSH-CTR is resistant due to the stateful nature of its decryption algorithm. In the two schemes SSH-CTRIV-CBC and SSH-EIV-CBC an attacker no longer sees the IV and therefore cannot deduce what changes have been induced in the target p_i^* when injecting c_i^* (c.f. equation (1)). It is therefore obvious that our attacks cannot be used to recover any plaintext when these schemes are used. However, this does not amount to a formal security proof for these schemes in the face of error-based side-channels of the type we have exploited in this paper.

6. Countermeasures

There are various actions which could be taken to make the SSH BPP resistant to our attacks, even with the continued presence of an encrypted length field, length checking and non-uniform error reporting.

In response to the vulnerability announcement concerning our attacks [8], OpenSSH was initially updated to make failure of the two block length checks more difficult to distinguish. The block length check (c.f. Section 2.1.2) in version 1.158 of the file `packet.c` no longer calls the function `fatal` but instead logs the error and then called the function `packet_disconnect`. This means that the error message caused by both the length check (c.f. Section 2.1.1) and the block length check are be the same and hence an attacker can no longer distinguish between these two events. This prevents our 14-bit plaintext recovery attack, but does not affect our attack recovering 32 bits. OpenSSH also issued a public response [14] to our attacks as reported in [8]. Subsequently, the OpenSSH team released OpenSSH 5.2. Unfortunately, we did not have time to analyse this new version with respect to our attacks before finalising this paper for publication. Our understanding is that OpenSSH 5.2 includes further countermeasures (in addition to uniform error reporting) to protect OpenSSH against our attacks.

Another countermeasure was suggested to us by Denis Bider of Bitvise [4]: simply randomise the length field if the length check fails. The system then proceeds with this new random length until an error is eventually sent when the MAC check fails. With this modification a length error and a MAC error are now indistinguishable and our attacks are no longer possible.

Note that moving the length checks so that they are executed after the MAC check, or removing the length checks altogether, only increases the success probability of our attacks to 1.

Another solution is to use one of the three schemes SSH-CTR, SSH-CTRIV-CBC and SSH-EIV-CBC proposed by Bellare *et al.*, since they resist our plaintext recovery attacks. An RFC already exists to standardise counter mode for

use in SSH [2] and AES in counter mode is supported by OpenSSH. A switch to AES in counter mode could most easily be enforced by limiting which encryption algorithms are offered during the ciphersuite negotiation that takes place as part of the SSH key exchange (see [26, Section 7.1]). We have been informed that not every SSH implementation supports counter mode, so that a switch to counter mode may cause backwards compatibility problems. OpenSSH 5.2 takes the approach of changing the default cipher order to prefer the AES CTR modes and the revised `arcfour256` cipher to CBC-mode ciphers. We are not aware of any support for SSH-CTRIV-CBC or SSH-EIV-CBC in SSH implementations.

A partial list of affected vendors and the countermeasures that they have adopted can be found at [22].

6.1. BPP Redesign

The kind of attacks that we have developed in this paper would seem to be a threat to almost any protocol that uses an encrypted length field, that runs over a transport protocol allowing the attacker to deliver small amounts of data at a time, and that reports errors. Given the problems that seem to be inherent in using an encrypted length field, we consider here how the BPP might be redesigned to avoid our attacks.

It is worth reiterating that the elimination of length checks on an encrypted length field only increases the success probability of our attacks. Moreover, replacing the current "encrypt and MAC" construction used in the BPP with an arbitrary authenticated encryption scheme may not eliminate our attacks. For example, consider the use of an encrypt-then-MAC scheme. This approach is known to be generically secure against chosen-ciphertext attacks [3] so would seem to be a good choice. But it is not hard to see that this construction would still be vulnerable to our attacks.

A more useful change to the BPP would be to no longer encrypt the 32-bit packet length field, but instead include it in the clear before the encrypted payload and protect it with the MAC. This would nullify our attacks, since now any length tests could not be made plaintext-revealing. However, making this change would reveal the length of BPP packets to an eavesdropper. It would also render SSH more vulnerable to DoS attacks, since the packet length field would still need to be checked before the MAC, but now the attacker would be able to manipulate the contents of this field at will (rather than in a probabilistic fashion). On the other hand, it could be argued that preventing DoS attacks should not be an overriding concern of a secure communications protocol such as SSH.

Another approach, suggested by a referee, would be to include an extra MAC, calculated on the length field, just after that field in the BPP protocol format.

7. Conclusions

We have presented plaintext-recovering attacks against SSH as implemented by OpenSSH. We have argued that the attacks arise from a fundamental design flaw in the RFC specifying the SSH BPP [26], and explained why the attacks are possible against a variant of the SSH BPP that was proven to be secure in [1]. We also described how schemes that are resistant to our plaintext recovery attacks may in some scenarios leak length information about the plaintext. We have also proposed countermeasures to the attacks.

Our attacks highlight the difficulties inherent in developing complex cryptographic constructions such as that employed in SSH, especially in view of the interactions that are possible between cryptographic processing and error reporting. At this point a quote from [1] seems appropriate:

> *"in the modern era of strong cryptography it would seem counterintuitive to voluntarily use a protocol with low security when it is possible to fix the security of SSH at low cost."*

While we see great value in the use of strong cryptography supported by arguments from the tradition of provable security, this quote raises the question: how do we know that making a change *does* fix the security of SSH? For example, our analysis shows that the proven-secure SSH-$NPC would be at least as vulnerable to our attacks as the current version of SSH.

In future work, we plan to examine formal models for the kinds of software side channels that seem to emerge in any reasonably complex implementation of cryptography. A start in this direction, in the specific context of padding oracle attacks, is provided in a recent paper of Paterson and Watson [16].

Acknowledgements

Martin Albrecht was supported by the Royal Holloway Valerie Myerscough Scholarship. Gaven Watson was supported by an EPSRC Industrial CASE studentship sponsored by BT Research Laboratories.

We thank the anonymous referees for their many constructive comments on the paper.

References

[1] M. Bellare, T. Kohno, and C. Namprempre. Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm. *ACM Transactions on Information and Systems Security*, 7(2):206–241, 2004.

[2] M. Bellare, T. Kohno, and C. Namprempre. The Secure Shell (SSH) Transport Layer Encryption Modes, RFC 4344, Jan. 2006. http://www.ietf.org/rfc/rfc4344.txt.

[3] M. Bellare and C. Namprempre. Authenticated Encryption: Relations Among Notions and Analysis of the Generic Composition Paradigm. In T. Okamoto, editor, *Proccedings of ASIACRYPT 2000*, LNCS 1976, pp. 531–545. Springer-Verlag, 2000.

[4] Denis Bider, personal communication, 18/11/2008.

[5] D. Bleichenbacher. Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1. In H. Krawczyk, editor, *Proccedings of CRYPTO 1998*, LNCS 1462, pp. 1–12. Springer-Verlag, 1998.

[6] D. Boneh and D. Brumley. Remote timing attacks are practical. In *Proceedings of the 12th Usenix Security Symposium*, 2003.

[7] B. Canvel, A.P. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password Interception in a SSL/TLS Channel. In D. Boneh, editor, *Proccedings of CRYPTO 2003*, LNCS 2729, pp. 583–599. Springer-Verlag, 2003.

[8] CPNI Vulnerability Advisory. Plaintext Recovery Attack Against SSH. http://www.cpni.gov.uk/Docs/Vulnerability_Advisory_SSH.txt, 14/11/2008 (revised 17/11/2008).

[9] W. Dai. An Attack Against SSH2 Protocol. Email to the SECSH Working Group available from ftp://ftp.ietf.org/ietf-mail-archive/secsh/2002-02.mail, 6th Feb. 2002.

[10] J.-P. Degabriele and K.G. Paterson. Attacking the IPsec Standards in Encryption-only Configurations. In *IEEE Symposium on Security and Privacy*, pp. 335–349, IEEE Computer Society, 2007.

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol, Version 1.1, RFC4346, April 2006. http://www.ietf.org/rfc/rfc4346.txt.

[12] A. Joux, G. Martinet and F. Valette. Blockwise-adaptive attackers: Revisiting the (in)security of some provably secure encryption models: CBC, GEM, IACBC. In Moti Yung, editor, *CRYPTO*, LNCS 2442, pp. 17–30, Springer-Verlag, 2002.

[13] OpenSSH Project, http://www.openssh.org/.

[14] OpenSSH Security Advisory: cbc.adv, http://www.openssh.com/txt/cbc.adv, 21/11/2008.

[15] OpenSSH Security, http://www.openssh.org/security.html.

[16] K.G. Paterson and G.J. Watson. Immunising CBC Mode Against Padding Oracle Attacks: A Formal Security Treatment. In R. Ostrovsky, R. De Prisco and I. Visconti, editors, *SCN 2008*, LNCS 5229, pp. 340–357, Springer-Verlag, 2008.

[17] K.G. Paterson and A.K.L. Yau. Cryptography in Theory and Practice: The Case of Encryption in IPsec. In S. Vaudenay, editor, *Eurocrypt 2006*, LNCS 4004, pp. 12–29, Springer-Verlag, 2006.

[18] Scapy Homepage, http://www.secdev.org/projects/scapy/.

[19] SSHFS Homepage, http://fuse.sourceforge.net/sshfs.html.

[20] SSH usage profiling, http://www.openssh.org/usage/index.html.

[21] D. Song, D. Wagner and X. Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In 10th USENIX Security Symposium, 2001, http://www.usenix.org/publications/library/proceedings/sec01/song.html.

[22] Unites States Computer Emergency Readiness Team (US-CERT). Vulnerability Note VU#958563 – SSH CBC vulnerability. http://www.kb.cert.org/vuls/id/958563, 24/11/2008 (revised 12/01/2009).

[23] S. Vaudenay. Security Flaws Induced by CBC Padding – Applications to SSL, IPSEC, WTLS .... In L.R. Knudsen, editor, *Proceedings of EUROCRYPT 2002*, LNCS 2332, pp. 534–546, Springer-Verlag, 2002.

[24] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture, RFC 4251, Jan. 2006. http://www.ietf.org/rfc/rfc4251.txt

[25] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol, RFC 4252, Jan. 2006. http://www.ietf.org/rfc/rfc4252.txt

[26] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol, RFC 4253, Jan. 2006. http://www.ietf.org/rfc/rfc4253.txt.

[27] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol, RFC 4254, Jan. 2006. http://www.ietf.org/rfc/rfc4254.txt.