DROWN: Breaking TLS using SSLv2

Nimrod Aviram, *Tel Aviv University;* **Sebastian Schinzel,** *Münster University of Applied Sciences;* **Juraj Somorovsky,** *Ruhr University Bochum;* **Nadia Heninger,** *University of Pennsylvania;* **Maik Dankel,** *Münster University of Applied Sciences;* **Jens Steube,** *Hashcat Project;* **Luke Valenta,** *University of Pennsylvania;* **David Adrian and J. Alex Halderman,** *University of Michigan;* **Viktor Dukhovni,** *Two Sigma and OpenSSL;* **Emilia Käsper,** *Google and OpenSSL;* **Shaanan Cohney,** *University of Pennsylvania;* **Susanne Engels and Christof Paar,** *Ruhr University Bochum;* **Yuval Shavitt,** *Tel Aviv University*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

DROWN: Breaking TLS using SSLv2

Nimrod Aviram[1], Sebastian Schinzel[2], Juraj Somorovsky[3], Nadia Heninger[4], Maik Dankel[2], Jens Steube[5], Luke Valenta[4], David Adrian[6], J. Alex Halderman[6], Viktor Dukhovni[7], Emilia Käsper[8], Shaanan Cohney[4], Susanne Engels[3], Christof Paar[3] and Yuval Shavitt[1]

[1]Department of Electrical Engineering, Tel Aviv University
[2]Münster University of Applied Sciences
[3]Horst Görtz Institute for IT Security, Ruhr University Bochum
[4]University of Pennsylvania
[5]Hashcat Project
[6]University of Michigan
[7]Two Sigma/OpenSSL
[8]Google/OpenSSL

Abstract

We present DROWN, a novel cross-protocol attack on TLS that uses a server supporting SSLv2 as an oracle to decrypt modern TLS connections.

We introduce two versions of the attack. The more general form exploits multiple unnoticed protocol flaws in SSLv2 to develop a new and stronger variant of the Bleichenbacher RSA padding-oracle attack. To decrypt a 2048-bit RSA TLS ciphertext, an attacker must observe 1,000 TLS handshakes, initiate 40,000 SSLv2 connections, and perform 2^{50} offline work. The victim client never initiates SSLv2 connections. We implemented the attack and can decrypt a TLS 1.2 handshake using 2048-bit RSA in under 8 hours, at a cost of $440 on Amazon EC2. Using Internet-wide scans, we find that 33% of all HTTPS servers and 22% of those with browser-trusted certificates are vulnerable to this protocol-level attack due to widespread key and certificate reuse.

For an even cheaper attack, we apply our new techniques together with a newly discovered vulnerability in OpenSSL that was present in releases from 1998 to early 2015. Given an unpatched SSLv2 server to use as an oracle, we can decrypt a TLS ciphertext in one minute on a single CPU—fast enough to enable man-in-the-middle attacks against modern browsers. We find that 26% of HTTPS servers are vulnerable to this attack.

We further observe that the QUIC protocol is vulnerable to a variant of our attack that allows an attacker to impersonate a server indefinitely after performing as few as 2^{17} SSLv2 connections and 2^{58} offline work.

We conclude that SSLv2 is not only weak, but actively harmful to the TLS ecosystem.

1 Introduction

TLS [13] is one of the main protocols responsible for transport security on the modern Internet. TLS and its precursor SSLv3 have been the target of a large number of cryptographic attacks in the research community, both on popular implementations and the protocol itself [33]. Prominent recent examples include attacks on outdated or deliberately weakened encryption in RC4 [3], RSA [5], and Diffie-Hellman [1], different side channels including Lucky13 [2], BEAST [14], and POODLE [35], and several attacks on invalid TLS protocol flows [5, 6, 12].

Comparatively little attention has been paid to the SSLv2 protocol, likely because the known attacks are so devastating and the protocol has long been considered obsolete. Wagner and Schneier wrote in 1996 that their attacks on SSLv2 "will be irrelevant in the long term when servers stop accepting SSL 2.0 connections" [41]. Most modern TLS clients do not support SSLv2 at all. Yet in 2016, our Internet-wide scans find that out of 36 million HTTPS servers, 6 million (17%) support SSLv2.

A Bleichenbacher attack on SSLv2. Bleichenbacher's padding oracle attack [8] is an adaptive chosen ciphertext attack against PKCS#1 v1.5, the RSA padding standard used in SSL and TLS. It enables decryption of RSA ciphertexts if a server distinguishes between correctly and incorrectly padded RSA plaintexts, and was termed the "million-message attack" upon its introduction in 1998, after the number of decryption queries needed to deduce a plaintext. All widely used SSL/TLS servers include countermeasures against Bleichenbacher attacks.

Our first result shows that the SSLv2 protocol is fatally vulnerable to a form of Bleichenbacher attack that enables

decryption of RSA ciphertexts. We develop a novel application of the attack that allows us to use a server that supports SSLv2 as an efficient padding oracle. This attack is a protocol-level flaw in SSLv2 that results in a feasible attack for 40-bit export cipher strengths, and in fact abuses the universally implemented countermeasures against Bleichenbacher attacks to obtain a decryption oracle.

We also discovered multiple implementation flaws in commonly deployed OpenSSL versions that allow an extremely efficient instantiation of this attack.

Using SSLv2 to break TLS. Second, we present a novel *cross-protocol attack* that allows an attacker to break a passively collected RSA key exchange for any TLS server if the RSA keys are also used for SSLv2, possibly on a different server. We call this attack DROWN (*Decrypting RSA using Obsolete and Weakened eNcryption*).

In its *general* version, the attack exploits the protocol flaws in SSLv2, does not rely on any particular library implementation, and is feasible to carry out in practice by taking advantage of commonly supported export-grade ciphers. In order to decrypt one TLS session, the attacker must passively capture about 1,000 TLS sessions using RSA key exchange, make 40,000 SSLv2 connections to the victim server, and perform 2^{50} symmetric encryption operations. We successfully carried out this attack using an optimized GPU implementation and were able to decrypt a 2048-bit RSA ciphertext in less than 18 hours on a GPU cluster and less than 8 hours using Amazon EC2.

We found that 11.5 million HTTPS servers (33%) are vulnerable to this attack, because many HTTPS servers that do not directly support SSLv2 share RSA keys with other services that do. Of servers offering HTTPS with browser-trusted certificates, 22% are vulnerable.

We also present a *special* version of DROWN that exploits flaws in OpenSSL for a more efficient oracle. It requires roughly the same number of captured TLS sessions as the general attack, but only half as many connections to the victim server and no large computations. This attack can be completed on a single core on commodity hardware in less than a minute, and is limited primarily by how fast the server can complete handshakes. It is fast enough that an attacker can perform man-in-the-middle attacks on live TLS sessions before the handshake times out, and downgrade a modern TLS client to RSA key exchange with a server that prefers non-RSA cipher suites. Our Internet-wide scans suggest that 79% of HTTPS servers that are vulnerable to the general attack, or 26% of all HTTPS servers, are also vulnerable to real-time attacks exploiting these implementation flaws.

Our results highlight the risk that continued support for SSLv2 imposes on the security of much more recent TLS versions. This is an instance of a more general phenomenon of insufficient domain separation, where older, vulnerable security standards can open the door to attacks on newer versions. We conclude that phasing out outdated and insecure standards should become a priority for standards designers and practitioners.

Disclosure. DROWN was assigned CVE-2016-0800. We disclosed our attacks to OpenSSL and worked with them to coordinate further disclosures. The specific OpenSSL vulnerabilities we discovered have been designated CVE-2015-3197, CVE-2016-0703, and CVE-2016-0704. In response to our findings, OpenSSL has made it impossible to configure a TLS server in such a way that it is vulnerable to DROWN. Microsoft had already disabled SSLv2 for all supported versions of IIS. We also disclosed the attack to the NSS developers, who have disabled SSLv2 on the last NSS tool that supported it and have hastened efforts to entirely remove the protocol from their codebase. In response to our disclosure, Google will disable QUIC support for non-whitelisted servers and modify the QUIC standard. We also notified IBM, Cisco, Amazon, the German CERT-Bund, and the Israeli CERT.

Online resources. Contact information, server test tools, and updates are available at https://drownattack.com.

2 Background

In the following, $a||b$ denotes concatenation of strings a and b. $a[i]$ references the i-th byte in a. (N,e) denotes an RSA public key, where N has byte-length ℓ_m ($|N| = \ell_m$) and e is the public exponent. The corresponding secret exponent is $d = 1/e \bmod \phi(N)$.

2.1 PKCS#1 v1.5 encryption padding

Our attacks rely on the structure of RSA PKCS#1 v1.5 padding. Although RSA PKCS#1 v2.0 implements OAEP, SSL/TLS still uses PKCS#1 v1.5. The PKCS#1 v1.5 encryption padding scheme [27] randomizes encryptions by prepending a random padding string PS to a message k (here, a symmetric session key) before RSA encryption:

1. The plaintext message is k, $\ell_k = |k|$. The encrypter generates a random byte string PS, where $|PS| \geq 8$, $|PS| = \ell_m - 3 - \ell_k$, and $\texttt{0x00} \notin \{PS[1], \ldots, PS[|PS|]\}$.
2. The encryption block is $m = 00||02||PS||00||k$.
3. The ciphertext is computed as $c = m^e \bmod N$.

To decrypt such a ciphertext, the decrypter first computes $m = c^d \bmod N$. Then it checks whether the decrypted message m is correctly formatted as a PKCS#1 v1.5-encoded message. We say that the ciphertext c and the decrypted message bytes $m[1]||m[2]||...||m[\ell_m]$ are PKCS#1 v1.5 conformant if:

$$m[1]||m[2] = \texttt{0x00}||\texttt{0x02}$$
$$\texttt{0x00} \notin \{m[3], \ldots, m[10]\}$$

If this condition holds, the decrypter searches for the first

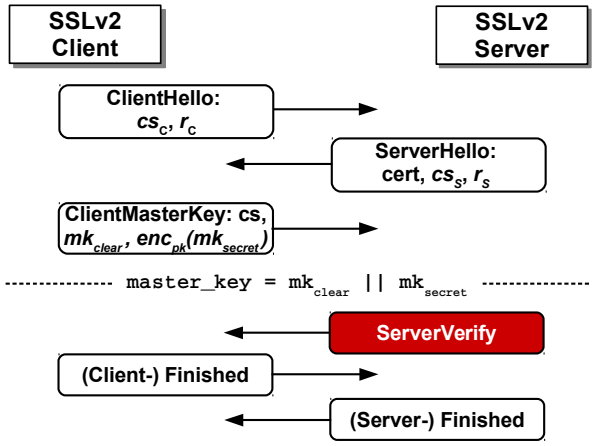

Figure 1: **SSLv2 handshake.** The server responds with a `ServerVerify` message directly after receiving an RSA-PKCS#1 v1.5 ciphertext contained in `ClientMasterKey`. This protocol feature enables our attack.

value $i > 10$ such that $m[i] = 0x00$. Then, it extracts $k = m[i+1]||\ldots||m[\ell_m]$. Otherwise, the ciphertext is rejected.

In SSLv3 and TLS, RSA PKCS#1 v1.5 is used to encapsulate the premaster secret exchanged during the handshake [13]. Thus, k is interpreted as the premaster secret. In SSLv2, RSA PKCS#1 v1.5 is used for encapsulation of an equivalent key denoted the `master_key`.

2.2 SSL and TLS

The first incarnation of the TLS protocol was the SSL (Secure Socket Layer) protocol, which was designed by Netscape in the 90s. The first two versions of SSL were immediately found to be vulnerable to trivial attacks [40, 41] which were fixed in SSLv3 [17]. Later versions of the standard were renamed TLS, and share a similar structure to SSLv3. The current version of the protocol is TLS 1.2; TLS 1.3 is currently under development.

An SSL/TLS protocol flow consists of two phases: handshake and application data exchange. In the first phase, the communicating parties agree on cryptographic algorithms and establish shared keys. In the second phase, these keys are used to protect the confidentiality and authenticity of the transmitted application data.

The handshake protocol was fundamentally redesigned in the SSLv3 version. This new handshake protocol was then used in later TLS versions up to TLS 1.2. In the following, we describe the RSA-based handshake protocols used in TLS and SSLv2, and highlight their differences.

The SSLv2 handshake protocol. The SSLv2 protocol description [22] is less formally specified than modern RFCs. Figure 1 depicts an SSLv2 handshake. A client initiates an SSLv2 handshake by sending a `ClientHello` message, which includes a list of cipher suites cs_c supported by the client and a client nonce r_c, termed `challenge`. The server responds with a `ServerHello` message, which contains a list of cipher suites cs_s supported by the server, the server certificate, and a server nonce r_s, termed `connection_ID`.

The client responds with a `ClientMasterKey` message, which specifies a cipher suite supported by both peers and key data used for constructing a `master_key`. In order to support *export* cipher suites with 40-bit security (e.g., `SSL_RC2_128_CBC_EXPORT40_WITH_MD5`), the key data is divided into two parts:

- mk_{clear}: A portion of the `master_key` sent in the `ClientMasterKey` message as plaintext (termed `clear_key_data` in the SSLv2 standard).
- mk_{secret}: A secret portion of the `master_key`, encrypted with RSA PKCS#1 v1.5 (termed `secret_key_data`).

The resulting `master_key` mk is constructed by concatenating these two keys: $mk = mk_{clear}||mk_{secret}$. For 40-bit export cipher suites, mk_{secret} is five bytes in length. For non-export cipher suites, the whole `master_key` is encrypted, and the length of mk_{clear} is zero.

The client and server can then compute session keys from the reconstructed `master_key` mk:

$$\texttt{server\_write\_key} = MD5(mk||\text{“0”}||r_c||r_s)$$
$$\texttt{client\_write\_key} = MD5(mk||\text{“1”}||r_c||r_s)$$

The server responds with a `ServerVerify` message consisting of the `challenge` r_c encrypted with the `server_write_key`. Both peers then exchange `Finished` messages in order to authenticate to each other.

Our attack exploits the fact that the server always decrypts an RSA-PKCS#1 v1.5 ciphertext, computes the `server_write_key`, and *immediately* responds with a `ServerVerify` message. The SSLv2 standard implies this message ordering, but does not make it explicit. However, we observed this behavior in every implementation we examined. Our attack also takes advantage of the fact that the encrypted mk_{secret} portion of the `master_key` can vary in length, and is only five bytes for export ciphers.

The TLS handshake protocol. In TLS [13] or SSLv3, the client initiates the handshake with a `ClientHello`, which contains a client random r_c and a list of supported cipher suites. The server chooses one of the cipher suites and responds with three messages, `ServerHello`, `Certificate`, and `ServerHelloDone`. These messages include the server's choice of cipher suite, server nonce r_s, and a server certificate with an RSA public key. The client then uses the public key to encrypt a newly generated 48-byte premaster secret *pms* and sends it to the server in a `ClientKeyExchange` message. The client and server then derive encryption and MAC keys from the premaster secret and the client and server random nonces. The details of this derivation are not important to our attack. The

client then sends `ChangeCipherSpec` and `Finished` messages. The `Finished` message authenticates all previous handshake messages using the derived keys. The server responds with its own `ChangeCipherSpec` and `Finished` messages.

The two main details relevant to our attacks are:

- The premaster secret is always 48 bytes long, independent of the chosen cipher suite. This is also true for export cipher suites.
- After receiving the `ClientKeyExchange` message, the server waits for the `ClientFinished` message, in order to authenticate the client.

2.3 Bleichenbacher's attack

Bleichenbacher's attack is a padding oracle attack—it exploits the fact that RSA ciphertexts should decrypt to PKCS#1 v1.5-compliant plaintexts. If an implementation receives an RSA ciphertext that decrypts to an invalid PKCS#1 v1.5 plaintext, it might naturally leak this information via an error message, by closing the connection, or by taking longer to process the error condition. This behavior can leak information about the plaintext that can be modeled as a cryptographic *oracle* for the decryption process. Bleichenbacher [8] demonstrated how such an oracle could be exploited to decrypt RSA ciphertexts.

Algorithm. In the simplest attack scenario, the attacker has a valid PKCS#1 v1.5 ciphertext c_0 that they wish to decrypt to discover the message m_0. They have no access to the private RSA key, but instead have access to an oracle $\mathcal{O}$ that will decrypt a ciphertext c and inform the attacker whether the most significant two bytes match the required value for a correct PKCS#1 v1.5 padding:

$$\mathcal{O}(c) = \begin{cases} 1 & \text{if } m = c^d \bmod N \text{ starts with } \texttt{0x00 02} \\ 0 & \text{otherwise.} \end{cases}$$

If the oracle answers with `1`, the attacker knows that $2B \le m \le 3B-1$, where $B = 2^{8(\ell_m - 2)}$. The attacker can take advantage of RSA malleability to generate new candidate ciphertexts for any s:

$$c = (c_0 \cdot s^e) \bmod N = (m_0 \cdot s)^e \bmod N$$

The attacker queries the oracle with c. If the oracle responds with 0, the attacker increments s and repeats the previous step. Otherwise, the attacker learns that for some r, $2B \le m_0 s - rN < 3B$. This allows the attacker to reduce the range of possible solutions to:

$$\frac{2B + rN}{s} \le m_0 < \frac{3B + rN}{s}$$

The attacker proceeds by refining guesses for s and r values and successively decreasing the size of the interval containing m_0. At some point the interval will contain a single valid value, m_0. Bleichenbacher's original paper describes this process in further detail [8].

Countermeasures. In order to protect against this attack, the decrypter must not leak information about the PKCS#1 v1.5 validity of the ciphertext. The ciphertext does not decrypt to a valid message, so the decrypter generates a fake plaintext and continues the protocol with this decoy. The attacker should not be able to distinguish the resulting computation from a correctly decrypted ciphertext.

In the case of SSL/TLS, the server generates a random premaster secret to continue the handshake if the decrypted ciphertext is invalid. The client will not possess the session key to send a valid `ClientFinished` message and the connection will terminate.

3 Breaking TLS with SSLv2

In this section, we describe our cross-protocol DROWN attack that uses an SSLv2 server as an oracle to efficiently decrypt TLS connections. The attacker learns the session key for targeted TLS connections but does not learn the server's private RSA key. We first describe our techniques using a generic SSLv2 oracle. In Section 4.1, we show how a protocol flaw in SSLv2 can be used to construct such an oracle, and describe our general DROWN attack. In Section 5, we show how an implementation flaw in common versions of OpenSSL leads to a more powerful oracle and describe our efficient special DROWN attack.

We consider a server accepting TLS connections from clients. The connections are established using a secure, state-of-the-art TLS version (1.0–1.2) and a `TLS_RSA` cipher suite with a private key unknown to the attacker.

The same RSA public key as the TLS connections is also used for SSLv2. For simplicity, our presentation will refer to the servers accepting TLS and SSLv2 connections as the same entity.

Our attacker is able to passively eavesdrop on traffic between the client and server and record RSA-based TLS traffic. The attacker may or may not be also required to perform active man-in-the-middle interference, as explained below.

The attacker can expect to decrypt one out of 1,000 intercepted TLS connections in our attack for typical parameters. This is a devastating threat in many scenarios. For example, a decrypted TLS connection might reveal a client's HTTP cookie or plaintext password, and an attacker would only need to successfully decrypt a single ciphertext to compromise the client's account. In order to collect 1,000 TLS connections, the attacker might simply wait patiently until sufficiently many connections are recorded. A less patient attacker might use man-in-the-middle interference, as in the BEAST attack [14].

3.1 A generic SSLv2 oracle

Our attacks make use of an oracle that can be queried on a ciphertext and leaks information about the decrypted plaintext; this abstractly models the information gained

from an SSLv2 server's behavior. Our SSLv2 oracles reveal many bytes of plaintext, enabling an efficient attack.

Our cryptographic oracle $\mathcal{O}$ has the following functionality: $\mathcal{O}$ decrypts an RSA ciphertext c and responds with ciphertext validity based on the decrypted message m. The ciphertext is valid only if m starts with `0x00 02` followed by non-null padding bytes, a delimiter byte `0x00`, and a `master_key` mk_{secret} of correct byte length ℓ_k. We call such a ciphertext *SSLv2 conformant*.

All of the SSLv2 padding oracles we instantiate give the attacker similar information about a PKCS#1 v1.5 conformant SSLv2 ciphertext:

$$\mathcal{O}(c) = \begin{cases} mk_{secret} & \text{if } c^d \bmod N = 00||02||PS||00||mk_{secret} \\ 0 & \text{otherwise.} \end{cases}$$

That is, the oracle $\mathcal{O}(c)$ will return the decrypted message mk_{secret} if it is queried on a PKCS#1 v1.5 conformant SSLv2 ciphertext c corresponding to a correctly PKCS#1 v1.5 padded encryption of mk_{secret}. The attacker then learns $\ell_k + 3$ bytes of $m = c^d \bmod N$: the first two bytes are $00||02$, and the last $\ell_k + 1$ bytes are $00||mk_{secret}$. The length ℓ_k of mk_{secret} varies based on the cipher suite used to instantiate the oracle. For export-grade cipher suites such as `SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5`, k will be 5 bytes, so the attacker learns 8 bytes of m.

3.2 DROWN attack template

Our attacker will use an SSLv2 oracle $\mathcal{O}$ to decrypt a TLS `ClientKeyExchange`. The behavior of $\mathcal{O}$ poses two problems for the attacker. First, a TLS key exchange ciphertext decrypts to a 48-byte premaster secret. But since no SSLv2 cipher suites have 48-byte key strengths, this means that a valid TLS ciphertext is invalid to our oracle $\mathcal{O}$. In order to apply Bleichenbacher's attack, the attacker must transform the TLS ciphertext into a valid SSLv2 key exchange message. Second, $\mathcal{O}$ is very restrictive, since it strictly checks the length of the unpadded message. According to Bardou et al. [4], Bleichenbacher's attack would require 12 million queries to such an oracle.[1]

Our attacker overcomes these problems by following this generic attack flow:

0. The attacker collects many encrypted TLS RSA key exchange messages.
1. The attacker converts one of the intercepted TLS ciphertexts containing a 48-byte premaster secret to an RSA PKCS#1 v1.5 encoded ciphertext valid to the SSLv2 oracle $\mathcal{O}$.
2. Once the attacker has obtained a valid SSLv2 RSA ciphertext, they can continue with a modified version of Bleichenbacher's attack, and decrypt the message after many more oracle queries.
3. The attacker then transforms the decrypted plaintext back into the original plaintext, which is one of the collected TLS handshakes.

We describe the algorithmic improvements we use to make each of these steps efficient below.

3.2.1 Finding an SSLv2 conformant ciphertext

The first step for the attacker is to transform the original TLS `ClientKeyExchange` message c_0 from a TLS conformant ciphertext into an SSLv2 conformant ciphertext.

For this task, we rely on the concept of *trimmers*, which were introduced by Bardou et al. [4]. Assume that the message $m_0 = {c_0}^d \bmod N$ is divisible by a small number t. In that case, $m_0 \cdot t^{-1} \bmod N$ simply equals the natural number m_0/t. If we choose $u \approx t$, and multiply the original message by $u \cdot t^{-1}$, the resulting number will lie near the original message: $m_0 \approx m_0/t \cdot u$.

This method gives a good chance of generating a new SSLv2 conformant message. Let c_0 be an intercepted TLS conformant RSA ciphertext, and let $m_0 = c_0^d \bmod N$ be the plaintext. We select a multiplier $s = u/t \bmod N = ut^{-1} \bmod N$ where u and t are coprime, compute the value $c_1 = c_0 s^e \bmod N$, and query $\mathcal{O}(c_1)$. We will receive a response if $m_1 = m_0 \cdot u/t$ is SSLv2 conformant.

As an example, let us assume a 2048-bit RSA ciphertext with $\ell_k = 5$, and consider the fraction $u = 7, t = 8$. The probability that $c_0 \cdot u/t$ will be SSLv2 conformant is 1/7,774, so we expect to make 7,774 oracle queries before obtaining a positive response from $\mathcal{O}$. Appendix A.1 gives more details on computing these probabilities.

3.2.2 Shifting known plaintext bytes

Once we have obtained an SSLv2 conformant ciphertext c_1, the oracle has also revealed the $\ell_k + 1$ least significant bytes (mk_{secret} together with the delimiter byte `0x00`) and two most significant `0x00 02` bytes of the SSLv2 conformant message m_1. We would like to *rotate* these known bytes around to the right, so that we have a large block of contiguous known most significant bytes of plaintext. In this section, we show that this can be accomplished by multiplying by some shift $2^{-r} \bmod N$. In other words, given an SSLv2 conformant ciphertext $c_1 = m_1^e \bmod N$, we can efficiently generate an SSLv2 conformant ciphertext $c_2 = m_2^e \bmod N$ where $m_2 = s \cdot m_1 \cdot 2^{-r} \bmod N$ and we know several most significant bytes of m_2.

Let $R = 2^{8(k+1)}$ and $B = 2^{8(\ell_m - 2)}$. Abusing notation slightly, let the integer $m_1 = 2 \cdot B + PS \cdot R + mk_{secret}$ be the plaintext satisfying $m_1^e = c_1 \bmod N$. At this stage, the ℓ_k-byte integer mk_{secret} is known and the $\ell_m - \ell_k - 3$-byte integer PS is not.

Let $\tilde{m_1} = 2 \cdot B + mk_{secret}$ be the known components of m_1, so $m_1 = \tilde{m_1} + PS \cdot R$. We can use this to compute a new plaintext for which we know many most significant

[1] See Table 1 in [4]. The oracle is denoted with the term `FFF`.

bytes. Consider the value:

$$m_1 \cdot R^{-1} \bmod N = \tilde{m_1} \cdot R^{-1} + PS \bmod N.$$

The value of PS is unknown and consists of $\ell_m - \ell_k - 3$ bytes. This means that the known value $\tilde{m_1} \cdot R^{-1}$ shares most of its $\ell_k + 3$ most significant bytes with $m_1 \cdot R^{-1}$.

Furthermore, we can iterate this process by finding a new multiplier s such that $m_2 = s \cdot m_1 \cdot R^{-1} \bmod N$ is also SSLv2 conformant. A randomly chosen $s < 2^{30}$ will work with probability $2^{-25.4}$. We can take use the bytes we have already learned about m_1 to efficiently compute such an s with only 678 oracle queries in expectation for a 2048-bit RSA modulus. Appendix A.3 gives more details.

3.2.3 Adapted Bleichenbacher iteration

It is feasible for all of our oracles to use the previous technique to entirely recover a plaintext message. However, for our SSLv2 protocol oracle it is cheaper after a few iterations to continue using Bleichenbacher's original attack. We can apply the original algorithm proposed by Bleichenbacher as described in Section 2.3.

Each step obtains a message that starts with the required `0x00 02` bytes after two queries in expectation. Since we know the value of the $\ell_k + 1$ least significant bytes after multiplying by any integer, we can query the oracle only on multipliers that cause the $(\ell_k + 1)$st least significant byte to be zero. However, we cannot ensure that the padding string is entirely nonzero; for a 2048-bit modulus this will hold with probability 0.37.

For a 2048-bit modulus, the total expected number of queries when using this technique to fully decrypt the plaintext is $2048 * 2/0.37 \approx 11,000$.

4 General DROWN

In this section, we describe how to use any correct SSLv2 implementation accepting export-grade cipher suites as a padding oracle. We then show how to adapt the techniques described in Section 3.2 to decrypt TLS RSA ciphertexts.

4.1 The SSLv2 export padding oracle

SSLv2 is vulnerable to a direct message side channel vulnerability exposing a Bleichenbacher oracle to the attacker. The vulnerability follows from three properties of SSLv2. First, the server immediately responds with a `ServerVerify` message after receiving the `ClientMasterKey` message, which includes the RSA ciphertext, without waiting for the `ClientFinished` message that proves the client knows the RSA plaintext. Second, when choosing 40-bit export RC2 or RC4 as the symmetric cipher, only 5 bytes of the `master_key` (mk_{secret}) are sent encrypted using RSA, and the remaining 11 bytes are sent in cleartext. Third, a server implementation that correctly implements the anti-Bleichenbacher countermeasure and receives an RSA key exchange message with invalid padding will generate a random premaster secret

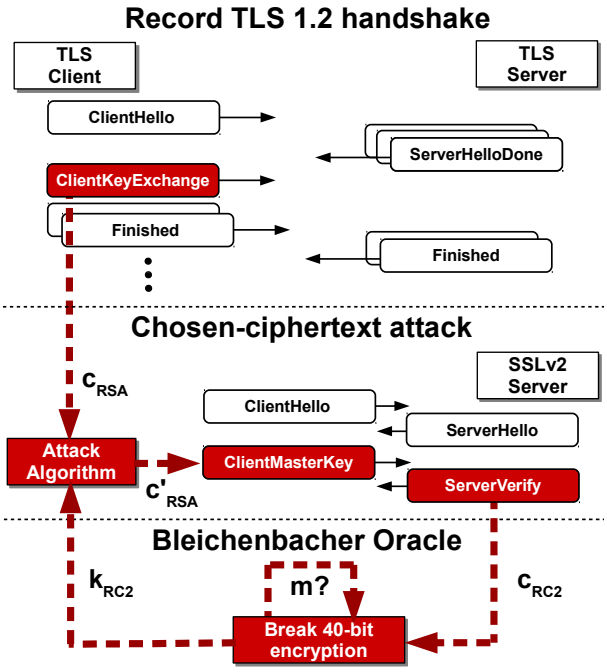

Figure 2: **SSLv2-based Bleichenbacher attack on TLS.** An attacker passively collects RSA ciphertexts from a TLS 1.2 handshake, and then performs oracle queries against a server that supports SSLv2 with the same public key to decrypt the TLS ciphertext.

and carry out the rest of the TLS handshake using this randomly generated key material.

This allows an attacker to deduce the validity of RSA ciphertexts in the following manner:

1. The attacker sends a `ClientMasterKey` message, which contains an RSA ciphertext c_0 and any choice of 11 clear key bytes for mk_{clear}. The server responds with a `ServerVerify` message, which contains the `challenge` encrypted using the `server_write_key`.

2. The attacker performs an *exhaustive search* over the possible values of the 5 bytes of the `master_key` mk_{secret}, computes the corresponding `server_write_key`, and checks whether the `ServerVerify` message decrypts to `challenge`. One value should pass this check; call it mk_0. Recall that if the RSA plaintext was valid, mk_0 is the unpadded data in the RSA plaintext c_0^d. Otherwise, mk_0 is a randomly generated sequence of 5 bytes.

3. The attacker re-connects to the server with the same RSA ciphertext c_0. The server responds with another `ServerVerify` message that contains the current `challenge` encrypted using the current `server_write_key`. If the decrypted RSA cipher-

text was valid, the attacker can use mk_0 to decrypt a correct `challenge` value from the `ServerVerify` message. Otherwise, if the `ServerVerify` message does not decrypt to `challenge`, the RSA ciphertext was invalid, and mk_0 must have been random.

Thus we can instantiate an oracle $\mathcal{O}_{\text{SSLv2-export}}$ using the procedure above; each oracle query requires two server connections and 2^{40} decryption attempts in the simplest case. For each oracle call $\mathcal{O}_{\text{SSLv2-export}}(c)$, the attacker learns whether c is valid, and if so, learns the two most significant bytes `0x00 02`, the sixth least significant `0x00` delimiter byte, and the value of the 5 least significant bytes of the plaintext m.

4.2 TLS decryption attack

In this section, we describe how the oracle described in Section 4.1 can be used to carry out a feasible attack to decrypt passively collected TLS ciphertexts.

As described in Section 3, we consider a server that accepts TLS connections from clients using an RSA public key that is exposed via SSLv2, and an attacker who is able to passively observe these connections.

We also assume the server supports export cipher suites for SSLv2. This can happen for two reasons. First, the same server operators that fail to follow best practices in disabling SSLv2 [40] may also fail to follow best practices by supporting export cipher suites. Alternatively, the server might be running a version of OpenSSL prior to January 2016, in which case it is vulnerable to the OpenSSL cipher suite selection bug described in Section 7, and an attacker may negotiate a cipher suite of his choice independent of the server configuration.

The attacker needs access to computing power sufficient to perform a 2^{50} time attack, mostly brute forcing symmetric key encryption. After our optimizations, this can be done with a one-time investment of a few thousand dollars of GPUs, or in a few hours for a few hundred dollars in the cloud. Our cost estimates are described in Section 4.3.

4.2.1 Constructing the attack

The attacker can exploit the SSLv2 vulnerability following the generic attack outline described in Section 3.2, consisting of several distinct phases:

0. The attacker passively collects 1,000 TLS handshakes from connections using RSA key exchange.
1. They then attempt to convert the intercepted TLS ciphertexts containing a 48-byte premaster secret to valid RSA PKCS#1 v1.5 encoded ciphertexts containing five-byte messages using the fractional trimmers described in Section 3.2.1, and querying $\mathcal{O}_{\text{SSLv2-export}}$. The attacker sends the modified ciphertexts to the server using fresh SSLv2 connections with weak symmetric ciphers and uses the `ServerVerify` messages to deduce ciphertext validity as described in the previous section. For each queried RSA ciphertext, the attacker must perform a brute force attack on the weak symmetric cipher. The attacker expects to obtain a valid SSLv2 ciphertext after roughly 10,000 oracle queries, or 20,000 connections to the server.
2. Once the attacker has obtained a valid SSLv2 RSA ciphertext $c_1 = m_1^e$, they use the shifting technique explained in Section 3.2.2 to find an integer s_1 such that $m_2 = m_1 \cdot 2^{-40} \cdot s_1$ is also SSLv2 conformant. Appendix A.4 contains more details on this step.
3. The attacker then applies the shifting technique again to find another integer s_2 such that $m_3 = m_2 \cdot 2^{-40} \cdot s_2$ is also SSLv2 conformant.
4. They then search for yet another integer s_3 such that $m_3 \cdot s_3$ is also SSLv2 conformant.
5. Finally, the attacker can continue with our adapted Bleichenbacher iteration technique described in Section 3.2.3, and decrypts the message after an expected 10,000 additional oracle queries, or 20,000 connections to the server.
6. The attacker can then transform the decrypted plaintext back into the original plaintext, which is one of the 1,000 intercepted TLS handshakes.

The rationale behind the different phases. Bleichenbacher's original algorithm requires a conformant message m_0, and a multiplier s_1 such that $m_1 = m_0 \cdot s_1$ is also conformant. Naïvely, it would appear we can apply the same algorithm here, after completing Phase 1. However, the original algorithm expects s_1 to be of size about 2^{24}. This is not the case when we use fractions for s_1, as the integer $s_1 = ut^{-1} \bmod N$ will be the same size as N.

Therefore, our approach is to find a conformant message for which we know the 5 most significant bytes; this will happen after multiple rotations and this message will be m_3. After finding such a message, finding s_3 such that $m_4 = m_3 \cdot s_3$ is also conformant becomes trivial. From there, we can finally apply the adapted Bleichenbacher iteration technique as described in Appendix A.5.

4.2.2 Attack performance

The attacker wishes to minimize three major costs in the attack: the number of recorded ciphertexts from the victim client, the number of connections to the victim server, and the number of symmetric keys to be brute forced. The requirements for each of these elements are governed by the set of fractions to be multiplied with each RSA ciphertext in the first phase, as described in Section 3.2.1.

Table 1 highlights a few choices for F and the resulting performance metrics for 2048-bit RSA keys. Appendix A provides more details on the derivation of these numbers

Optimizing for	Cipher-texts	$\|F\|$	SSLv2 connections	Offline work
offline work	12,743	1	50,421	$2^{49.64}$
offline work	1,055	10	46,042	$2^{50.63}$
compromise	4,036	2	41,081	$2^{49.98}$
online work	2,321	3	38,866	$2^{51.99}$
online work	906	8	39,437	$2^{52.25}$

Table 1: **2048-bit Bleichenbacher attack complexity.** The cost to decrypt one ciphertext can be adjusted by choosing the set of fractions F the attacker applies to each of the passively collected ciphertexts in the first step of the attack. This choice affects several parameters: the number of these collected ciphertexts, the number of connections the attacker makes to the SSLv2 server, and the number of offline decryption operations.

Key size	Phase 1	Phases 2–5	Total queries	Offline work
1024	4,129	4,132	8,261	$2^{50.01}$
2048	6,919	12,468	19,387	$2^{50.76}$
4096	18,286	62,185	80,471	$2^{52.16}$

Table 2: **Oracle queries required by our attack.** In Phase 1, the attacker queries the oracle until an SSLv2 conformant ciphertext is found. In Phases 2–5, the attacker decrypts this ciphertext using leaked plaintext. These numbers minimize total queries. In our attack, an oracle query represents two server connections.

and other optimization choices. Table 2 gives the expected number of Bleichenbacher queries for different RSA key sizes, when minimizing total oracle queries.

4.3 Implementing general DROWN with GPUs

The most computationally expensive part of our general DROWN attack is breaking the 40-bit symmetric key, so we developed a highly optimized GPU implementation of this brute force attack. Our first naïve GPU implementation performed around 26MH/s, where MH denotes the time required for testing one million possible values of mk_{secret}. Our optimized implementation runs at a final speed of 515MH/s, a speedup factor of 19.8.

We obtained our improvements through a number of optimizations. For example, our original implementation ran into a communication bottleneck in the PCI-E bus in transmitting candidate keys from CPU to GPU, so we removed this bottleneck by generating key candidates on the GPU itself. We optimized memory management, including storing candidate keys and the RC2 permutation table in constant memory, which is almost as fast as a register, instead of slow global memory.

We experimentally evaluated our optimized implementation on a local cluster and in the cloud. We used it to execute a full attack of $2^{49.6}$ tested keys on each platform. The required number of keys to test during the attack is a random variable, distributed geometrically, with an expectation that ranges between $2^{49.6}$ and $2^{52.5}$ depending on the choice of optimization parameters. We treat a full attack as requiring $2^{49.6}$ tested keys overall.

Hashcat. Hashcat [20] is an open source optimized password-recovery tool. The Hashcat developers allowed us to use their GPU servers for our attack evaluation. The servers contain a total of 40 GPUs: 32 Nvidia GTX 980 cards, and 8 AMD R9 290X cards. The value of this equipment is roughly $18,040. Our full attack took less than 18 hours to complete on the Hashcat servers, with the longest single instance taking 17h9m.

Amazon EC2. We also ran our optimized GPU code on the Amazon Elastic Compute Cloud (EC2) service. We used a cluster composed of 200 variable-price "spot" instances: 150 `g2.2xlarge` instances, each containing one high-performance NVIDIA GPU with 1,536 CUDA cores and 50 `g2.8xlarge` instances, each containing four of these GPUs. When we ran our experiments in January 2016, the average spot rates we paid were $0.09/hr and $0.83/hr respectively. Our full attack finished in under 8 hours including startup and shutdown for a cost of $440.

4.4 OpenSSL SSLv2 cipher suite selection bug

General DROWN is a protocol flaw, but the population of vulnerable hosts is increased due to a bug in OpenSSL that causes many servers to erroneously support SSLv2 and export ciphers even when configured not to. The OpenSSL team intended to disable SSLv2 by default in 2010, with a change that removed all SSLv2 cipher suites from the default list of ciphers offered by the server [36]. However, the code for the protocol itself was not removed in standard builds and SSLv2 itself remained enabled. We discovered a bug in OpenSSL's SSLv2 cipher suite negotiation logic that allows clients to select SSLv2 cipher suites even when they are not explicitly offered by the server. We notified the OpenSSL team of this vulnerability, which was assigned CVE-2015-3197. The problem was fixed in OpenSSL releases 1.0.2f and 1.0.1r [36].

5 Special DROWN

We discovered multiple vulnerabilities in recent (but not current) versions of the OpenSSL SSLv2 handshake code that create even more powerful Bleichenbacher oracles, and drastically reduce the amount of computation required to implement our attacks. The vulnerabilities, designated CVE-2016-0703 and CVE-2016-0704, were present in the OpenSSL codebase from at least the start of the repository, in 1998, until they were unknowingly fixed on March

4, 2015 by a patch [28] designed to correct an unrelated problem [11]. By adapting DROWN to exploit this special case, we can significantly cut both the number of connections and the computational work required.

5.1 The OpenSSL "extra clear" oracle

Prior to the fix, OpenSSL servers improperly allowed the `ClientMasterKey` message to contain `clear_key_data` bytes for *non-export* ciphers. When such bytes are present, the server substitutes them for bytes from the encrypted key. For example, consider the case that the client chooses a 128-bit cipher and sends a 16-byte encrypted key $k[1], k[2], \ldots, k[16]$ but, contrary to the protocol specification, includes 4 null bytes of `clear_key_data`. Vulnerable OpenSSL versions will construct the following `master_key`:

$$[00\ 00\ 00\ 00\ k[1]\ k[2]\ k[3]\ k[4]\ \ldots\ k[9]\ k[10]\ k[11]\ k[12]]$$

This enables a straightforward key recovery attack against such versions. An attacker that has intercepted an SSLv2 connection takes the RSA ciphertext of the encrypted key and replays it in non-export handshakes to the server with varying lengths of `clear_key_data`. For a 16-byte encrypted key, the attacker starts with 15 bytes of clear key, causing the server to use the `master_key`:

$$[00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ k[1]]$$

The attacker can brute force the first byte of the encrypted key by finding the matching `ServerVerify` message among 256 possibilities. Knowing $k[1]$, the attacker makes another connection with the same RSA ciphertext but 14 bytes of clear key, resulting in the `master_key`:

$$[00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ k[1]\ k[2]]$$

The attacker can now easily brute force $k[2]$. With only 15 probe connections and an expected $15 \cdot 128 = 1,920$ trial encryptions, the attacker learns the entire `master_key` for the recorded session.

As this oracle is obtained by improperly sending unexpected clear-key bytes, we call it the Extra Clear oracle.

This session key-recovery attack can be directly converted to a Bleichenbacher oracle. Given a candidate ciphertext and symmetric key length ℓ_k, the attacker sends the ciphertext with ℓ_k known bytes of `clear_key_data`. The oracle decision is simple:

- If the ciphertext is valid, the `ServerVerify` message will reflect a `master_key` consisting of those ℓ_k known bytes.
- If the ciphertext is invalid, the `master_key` will be replaced with ℓ_k random bytes (by following the countermeasure against the Bleichenbacher attack), resulting in a different `ServerVerify` message.

This oracle decision requires one connection to the server and one `ServerVerify` computation. After the attacker has found a valid ciphertext corresponding to a ℓ_k-byte encrypted key, they recover the ℓ_k plaintext bytes by repeating the key recovery attack from above. Thus our oracle $\mathcal{O}_{\text{SSLv2-extra-clear}}(c)$ requires one connection to determine whether c is valid. After ℓ_k connections, the attacker additionally learns the ℓ_k least significant bytes of m. We model this as a single oracle call, but the number of server connections will vary depending on the response.

5.2 MITM attack against TLS

Special DROWN is fast enough that it can decrypt a TLS premaster secret *online*, during a connection handshake. A man-in-the-middle attacker can use it to compromise connections between modern browsers and TLS servers—even those configured to prefer non-RSA cipher suites.

The MITM attacker impersonates the server and sends a `ServerHello` message that selects a cipher suite with RSA as the key-exchange method. Then, the attacker uses special DROWN to decrypt the premaster secret. The main difficulty is completing the decryption and producing a valid `ServerFinished` message before the client's connection times out. Most browsers will allow the handshake to last up to one minute [1].

The attack requires targeting an average of 100 connections, only one of which will be attacked, probabilistically. The simplest way for the attacker to facilitate this is to use JavaScript to cause the client to connect repeatedly to the victim server, as described in Section 3. Each connection is tested against the oracle with only small number of fractions, and the attacker can discern immediately when he receives a positive response from the oracle.

Note that since the decryption must be completed online, the Leaky Export oracle cannot be used, and the attack uses only the Extra Clear oracle.

5.2.1 Constructing the attack

We will use `SSL_DES_192_EDE3_CBC_WITH_MD5` as the cipher suite, allowing the attacker to recover 24 bytes of key at a time. The attack works as follows:

0. The attacker causes the victim client to connect repeatedly to the victim server, with at least 100 connections.
1. The attacker uses the fractional trimmers as described in Section 3.2.1 to convert one of the TLS ciphertexts into an SSLv2 conformant ciphertext c_0.
2. Once the attacker has obtained a valid SSLv2 ciphertext c_1, they repeatedly use the shifting technique described in Section 3.2.2 to rotate the message by 25 bytes each iteration, learning 27 bytes with each shift. After several iterations, they have learned the entire plaintext.
3. The attacker then transforms the decrypted SSLv2 plaintext into the decrypted TLS plaintext.

Using 100 fractional trimmers, this more efficient oracle attack allows the attacker to recover one in 100 TLS session keys using only about 27,000 connections to the server, as described in Appendix A.6. The computation cost is so low that we can complete the full attack on a single workstation in under one minute.

5.3 The OpenSSL "leaky export" oracle

In addition to the extra clear implementation bug, the same set of OpenSSL versions also contain a separate bug, where they do not follow the correct algorithm for their implementation of the Bleichenbacher countermeasure. We now describe this faulty implementation:

- The SSLv2 `ClientKeyExchange` message contains the mk_{clear} bytes immediately before the ciphertext c. Let p be the buffer starting at the first mk_{clear} byte.
- Decrypt c in place. If the decryption operation succeeds, and c decrypted to a plaintext of a correct padded length, p now contains the 11 mk_{clear} bytes followed by the 5 mk_{secret} bytes.
- If c decrypted to an unpadded plaintext k of incorrect length, the decryption operation overwrites the first $j = min(|k|, 5)$ bytes of c with the first j bytes of k.
- If c is not SSLv2 conformant and the decryption operation failed, randomize the first five bytes of p, which are the first five bytes of mk_{clear}.

This behavior allows the attacker to distinguish between these three cases. Suppose the attacker sends 11 null bytes as mk_{clear}. Then these are the possible cases:

1. c decrypts to a correctly padded plaintext k of the expected length, 5 bytes. Then the following `master_key` will be constructed:

   $[00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ k[1]\ k[2]\ k[3]\ k[4]\ k[5]]$

2. c decrypts to a correctly padded plaintext k of a wrong length. Let r be the five random bytes the server generated. The yielded `master_key` will be:

   $[r[1]\ r[2]\ r[3]\ r[4]\ r[5]\ 00\ 00\ 00\ 00\ 00\ 00\ k[1]\ k[2]\ k[3]\ k[4]\ k[5]]$

   when $|k| \geq 5$. If $|k| < 5$, the server substitutes the first $|k|$ bytes of c with the first $|k|$ bytes of k. Using $|k| = 3$ as an example, the `master_key` will be:

   $[r[1]\ r[2]\ r[3]\ r[4]\ r[5]\ 00\ 00\ 00\ 00\ 00\ 00\ k[1]\ k[2]\ k[3]\ c[4]\ c[5]]$

3. c is not SSLv2 conformant, and hence the decryption operation failed. The resulting `master_key` will be:

   $[r[1]\ r[2]\ r[3]\ r[4]\ r[5]\ 00\ 00\ 00\ 00\ 00\ 00\ c[1]\ c[2]\ c[3]\ c[4]\ c[5]]$

The attacker detects case (3) by performing an exhaustive search over the 2^{40} possibilities for r, and checking whether any of the resulting values for the `master_key` correctly decrypts the observed `ServerVerify` message. If no r value satisfies this property, then c^d starts with bytes `0x00 02`. The attacker then distinguishes between cases (1) and (2) by performing an exhaustive search over the five bytes of k, and checking whether any of the resulting values for mk correctly decrypts the observed `ServerVerify` message.

As this oracle leaks information when using export ciphers, we have named it the Leaky Export oracle.

In conclusion, $\mathcal{O}_{\text{SSLv2-export-leaky}}$ allows an attacker to obtain a valid oracle response for all ciphertexts which decrypt to a correctly-padded plaintext of *any* length. This is in contrary to the previous oracles $\mathcal{O}_{\text{SSLv2-extra-clear}}$ and $\mathcal{O}_{\text{SSLv2-export}}$, which required the plaintext to be of a specific length. Each oracle query to $\mathcal{O}_{\text{SSLv2-export-leaky}}$ requires one connection to the server and 2^{41} offline work.

Combining the two oracles. The attacker can use the Extra Clear and Leaky Export oracles together in order to reduce the number of queries required for the TLS decryption attack. They first test a TLS conformant ciphertext for divisors using the Leaky Export oracle, then use fractions dividing the plaintext with both oracles. Once the attacker has obtained a valid SSLv2 ciphertext c_1, they repeatedly use the shifting technique described in Section 3.2.2 to rotate the message by 25 bytes each iteration while choosing 3DES as the symmetric cipher, learning 27 bytes with each shift. After several iterations, they have learned the entire plaintext, using 6,300 queries (again for a 2048-bit modulus). This brings the overall number of queries for this variant of the attack to $900 + 16 * 4 + 6,300 = 7,264$. These parameter choices are not necessarily optimal. We give more details in Appendix A.7.

6 Extending the attack to QUIC

DROWN can also be extended to a feasible-time man-in-the-middle attack against QUIC [26]. QUIC [10, 39] is a recent cryptographic protocol designed and implemented by Google that is intended to reduce the setup time to establish a secure connection while providing security guarantees analogous to TLS. QUIC's security relies on a static "server config" message signed by the server's public key. Jager et al. [26] observe that an attacker who can forge a signature on a malicious QUIC server config once would be able to impersonate the server indefinitely. In this section, we show an attacker with significant resources would be able to mount such an attack against a server whose RSA public keys is exposed via SSLv2.

A QUIC client receives a "server config" message, signed by the server's public key, which enumerates connection parameters: a static elliptic curve Diffie-Hellman public value, and a validity period. In order to mount a man-in-the-middle attack against any client, the attacker wishes to generate a valid server config message containing their own Diffie-Hellman value, and an expiration date far in the future.

The attacker needs to present a forged QUIC config to the client in order to carry out a successful attack. This is

Protocol	Attack type	Oracle	SSLv2 connections	Offline work	See §
TLS	Decrypt	SSLv2	41,081	2^{50}	4.2
TLS	Decrypt	Special	7,264	2^{51}	5.3
TLS	MITM	Special	27,000	2^{15}	5.2
QUIC	MITM	SSLv2	2^{25}	2^{65}	6.1
QUIC	MITM	Special	2^{25}	2^{25}	6.2
QUIC	MITM	Special	2^{17}	2^{58}	6.2

Table 3: **Summary of attacks.** "Oracle" denotes the oracle required to mount each attack, which also implies the vulnerable set of SSLv2 implementations. SSLv2 denotes any SSLv2 implementation, while "Special" denotes an OpenSSL version vulnerable to special DROWN.

straightforward, since QUIC discovery may happen over non-encrypted HTTP [19]. The server does not even need to support QUIC at all: an attacker could impersonate the attacked server over an unencrypted HTTP connection and falsely indicate that the server supports QUIC. The next time the client connects to the server, it will attempt to connect using QUIC, allowing the attacker to present the forged "server config" message and execute the attack [26].

6.1 QUIC signature forgery attack based on general DROWN

The attack proceeds much as in Section 3.2, except that some of the optimizations are no longer applicable, making the attack more expensive.

The first step is to discover a valid, PKCS conformant SSLv2 ciphertext. In the case of TLS decryption, the input ciphertext was PKCS conformant to begin with; this is not the case for the QUIC message c_0. Thus for the first phase, the attacker iterates through possible multiplier values s until they randomly encounter a valid SSLv2 message in $c_0 \cdot s^d$. For 2048-bit RSA keys, the probability of this random event is $P_{rnd} \approx 2^{-25}$; see Section 3.2.

Once the first SSLv2 conformant message is found, the attacker proceeds with the signature forgery as they would in Step 2 of the TLS decryption attack. The required number of oracle queries for this step is roughly 12,468 for 2048-bit RSA keys.

Attack cost. The overall oracle query cost is dominated by the $2^{25} \approx 34$ million expected queries in the first phase, above. At a rate of 388 queries/second, the attacker would finish in one day; at a rate of 12 queries/second they would finish in one month.

For the SSLv2 export padding oracle, the offline computation to break a 40-bit symmetric key for each query requires iterating over 2^{65} keys. At our optimized GPU implementation rate of 515 million keys per second, this would require 829,142 GPU days. Our experimental GPU hardware retails for $400. An investment of $10 million to purchase 25,000 GPUs would reduce the wall clock time for the attack to 33 days.

Our implementation run on Amazon EC2 processed about 174 billion keys per `g2.2xlarge` instance-hour, so at a cost of $0.09/instance-hour the full attack would cost $9.5 million and could be parallelized to Amazon's capacity.

6.2 Optimized QUIC signature forgery based on special DROWN

For targeted servers that are vulnerable to special DROWN, we are unaware of a way to combine the two special DROWN oracles; the attacker would have to choose a single oracle which minimizes his subjective cost. For the Extra Clear oracle, there is only negligible computation per oracle query, so the computational cost for the first phase is 2^{25}. For the Leaky Export oracle, as explained below, the required offline work is 2^{58}, and the required number of server connections is roughly 145,573. Both oracles appear to bring this attack well within the means of a moderately provisioned adversary.

Mounting the attack using Leaky Export. For a 2048-bit RSA modulus, the probability of a random message being conformant when querying $\mathcal{O}_{\text{SSLv2-export-leaky}}$ is $P_{rnd} \approx (1/256)^2 * (255/256)^8 * (1-(255/256)^{246}) \approx 2^{-17}$. Therefore, to compute c^d when c is not SSLv2 conformant, the attacker randomly generates values for s and tests $c \cdot s^e$ against the Leaky Export oracle. After roughly $2^{17} \approx 131{,}000$ queries, they obtain a positive response, and learn that $c^d \cdot s$ starts with bytes `0x00 02`.

Naïvely, it would seem the attacker can then apply one of the techniques presented in this work, but $\mathcal{O}_{\text{SSLv2-export-leaky}}$ does not provide knowledge of any least significant plaintext bytes when the plaintext length is not at most the correct one. Instead, the attacker proceeds directly according to the algorithm presented in [4]. Referring to Table 1 in [4], $\mathcal{O}_{\text{SSLv2-export-leaky}}$ is denoted with the term FFT, as it returns a positive response for a correctly padded plaintext of any length, and the median number of required queries for this oracle is 14,501. This number of queries is dominated by the 131,000 queries the attacker has already executed. As each query requires testing roughly 2^{41} keys, the required offline work is approximately 2^{58}.

Future changes to QUIC. In addition to disabling QUIC support for non-whitelisted servers, Google have informed us that they plan to change the QUIC standard, so that the "server config" message will include a client nonce to prove freshness. They also plan to limit QUIC discovery to HTTPS.

Protocol	Port	*All certificate*			*Trusted certificates*		
		SSL/TLS	**SSLv2 support**	**Vulnerable key**	**SSL/TLS**	**SSLv2 support**	**Vulnerable key**
SMTP	25	3,357 K	936 K (28%)	1,666 K (50%)	1,083 K	190 K (18%)	686 K (63%)
POP3	110	4,193 K	404 K (10%)	1,764 K (42%)	1,787 K	230 K (13%)	1,031 K (58%)
IMAP	143	4,202 K	473 K (11%)	1,759 K (42%)	1,781 K	223 K (13%)	1,022 K (57%)
HTTPS	443	34,727 K	5,975 K (17%)	11,444 K (33%)	17,490 K	1,749 K (10%)	3,931 K (22%)
SMTPS	465	3,596 K	291 K (8%)	1,439 K (40%)	1,641 K	40 K (2%)	949 K (58%)
SMTP	587	3,507 K	423 K (12%)	1,464 K (42%)	1,657 K	133 K (8%)	986 K (59%)
IMAPS	993	4,315 K	853 K (20%)	1,835 K (43%)	1,909 K	260 K (14%)	1,119 K (59%)
POP3S	995	4,322 K	884 K (20%)	1,919 K (44%)	1,974 K	304 K (15%)	1,191 K (60%)
(Alexa Top 1M)	443	611 K	82 K (13%)	152 K (25%)	456 K	38 K (8%)	109 K (24%)

Table 4: **Hosts vulnerable to general DROWN.** We performed Internet-wide scans to measure the number of hosts supporting SSLv2 on several different protocols. A host is vulnerable to DROWN if its public key is exposed anywhere via SSLv2. Overall vulnerability to DROWN is much larger than support for SSLv2 due to widespread reuse of keys.

7 Measurements

We performed Internet-wide scans to analyze the number of systems vulnerable to DROWN. A host is directly vulnerable to general DROWN if it supports SSLv2. Similarly, a host is directly vulnerable to special DROWN if it supports SSLv2 and has the extra clear bug (which also implies the leaky export bug). These directly vulnerable hosts can be used as oracles to attack any other host with the same key. Hosts that do not support SSLv2 are still vulnerable to general or special DROWN if their RSA key pair is exposed by any general or special DROWN oracle, respectively. The oracles may be on an entirely different host or port. Additionally, any host serving a browser-trusted certificate is vulnerable to a special DROWN man-in-the-middle if any name on the certificate appears on any other certificate containing a key that is exposed by a special DROWN oracle.

We used ZMap [16] to perform full IPv4 scans on eight different ports during late January and February 2016. We examined port 443 (HTTPS), and common email ports 25 (SMTP with STARTTLS), 110 (POP3 with STARTTLS), 143 (IMAP with STARTTLS), 465 (SMTPS), 587 (SMTP with STARTTLS), 993 (IMAPS), and 995 (POP3S). For each open port, we attempted three complete handshakes: one normal handshake with the highest available SSL/TLS version; one SSLv2 handshake requesting an export RC2 cipher suite; and one SSLv2 handshake with a non-export cipher and sixteen bytes of plaintext key material sent during key exchange, which we used to detect if a host has the extra clear bug.

We summarize our general DROWN results in Table 4. The fraction of SSL/TLS hosts that directly supported SSLv2 varied substantially across ports. 28% of SMTP servers on port 25 supported SSLv2, likely due to the opportunistic encryption model for email transit. Since SMTP fails-open to plaintext, many servers are configured with support for the largest possible set of protocol versions and cipher suites, under the assumption that even bad or obsolete encryption is better than plaintext [9]. The other email ports ranged from 8% for SMTPS to 20% for POP3S and IMAPS. We found 17% of all HTTPS servers, and 10% of those with a browser-trusted certificate, are directly vulnerable to general DROWN.

OpenSSL SSLv2 cipher suite selection bug. We discovered that OpenSSL servers do not respect the cipher suites advertised in the SSLv2 `ServerHello` message. That is, a malicious client can select an *arbitrary* cipher suite in the `ClientMasterKey` message, regardless of the contents of the `ServerHello`, and force the use of export cipher suites even if they are explicitly disabled in the server configuration. To fully detect SSLv2 oracles, we configured our scanner to ignore the `ServerHello` cipher list. The cipher selection bug helps explain the wide support for SSLv2—the protocol appeared disabled, but non-standard clients could still complete handshakes.

Widespread public key reuse. Reuse of RSA key material across hosts and certificates is widespread [21, 23]. Often this is benign: organizations may issue multiple TLS certificates for distinct domains with the same public key in order to simplify use of TLS acceleration hardware and load balancing. However, there is also evidence that system administrators may not entirely understand the role of the public key in certificates. For example, in the wake of the Heartbleed vulnerability, a substantial fraction of compromised certificates were reissued with the same public key [15]. The number of hosts vulnerable to DROWN rises significantly when we take RSA key reuse into account. For HTTPS, 17% of hosts are vulnerable to general DROWN because they support both TLS and SSLv2 on the HTTPS port, but 33% are vulnerable when considering RSA keys used by another service.

Protocol	Port	*Any certificate*			*Trusted certificates*		
		SSL/TLS	**Special DROWN oracles**	**Vulnerable key**	**SSL/TLS**	**Vulnerable key**	**Vulnerable name**
SMTP	25	3,357 K	855 K (25%)	896 K (27%)	1,083 K	305 K (28%)	398 K (37%)
POP3	110	4,193 K	397 K (9%)	946 K (23%)	1,787 K	485 K (27%)	674 K (38%)
IMAP	143	4,202 K	457 K (11%)	969 K (23%)	1,781 K	498 K (30%)	690 K (39%)
HTTPS	443	34,727 K	4,029 K (12%)	9,089 K (26%)	17,490 K	2,523 K (14%)	3,793 K (22%)
SMTPS	465	3,596 K	334 K (9%)	765 K (21%)	1,641 K	430 K (26%)	630 K (38%)
SMTP	587	3,507 K	345 K (10%)	792 K (23%)	1,657 K	482 K (29%)	667 K (40%)
IMAPS	993	4,315 K	892 K (21%)	1,073 K (25%)	1,909 K	602 K (32%)	792 K (42%)
POP3S	995	4,322 K	897 K (21%)	1,108 K (26%)	1,974 K	641 K (32%)	835 K (42%)
(Alexa Top 1M)	443	611 K	22 K (4%)	52 K (9%)	456 K	33 K (7%)	85 K (19%)

Table 5: **Hosts vulnerable to special DROWN.** A server is vulnerable to special DROWN if its key is exposed by a host with the CVE-2016-0703 bug. Since the attack is fast enough to enable man-in-the-middle attacks, a server is also vulnerable (to impersonation) if any name in its certificate is found in any trusted certificate with an exposed key.

Special DROWN. As shown in Table 5, 9.1 M HTTPS servers (26%) are vulnerable to special DROWN, as are 2.5 M HTTPS servers with browser-trusted certificates (14%). 66% as many HTTPS hosts are vulnerable to special DROWN as to general DROWN (70% for browser-trusted servers). While 2.7 M public keys are vulnerable to general DROWN, only 1.1 M are vulnerable to special DROWN (41% as many). Vulnerability among Alexa Top Million domains is also lower, with only 9% of domains vulnerable (7% for browser-trusted domains).

Since special DROWN enables active man-in-the-middle attacks, any host serving a browser-trusted certificate with at least one name that appears on any certificate with an RSA key exposed by a special DROWN oracle is vulnerable to an impersonation attack. Extending our search to account for certificates with shared names, we find that 3.8 M (22%) hosts with browser-trusted certificates are vulnerable to man-in-the-middle attacks, as well as 19% of the browser-trusted domains in the Alexa Top Million.

8 Related work

TLS has had a long history of implementation flaws and protocol attacks [2,3,7,14,15,35,38]. We discuss relevant Bleichenbacher and cross-protocol attacks below.

Bleichenbacher's attack. Bleichenbacher's adaptive chosen ciphertext attack against SSL was first published in 1998 [8]. Several works have adapted his attack to different scenarios [4, 25, 29]. The TLS standard explicitly introduces countermeasures against the attack [13], but several modern implementations have been discovered to be vulnerable to timing-attack variants in recent years [34, 42]. These side-channel attacks are implementation failures and only apply when the attacker is co-located with the victim.

Cross-protocol attacks. Jager et al. [26] showed that a cross-protocol Bleichenbacher RSA padding oracle attack is possible against the proposed TLS 1.3 standard, in spite of the fact that TLS 1.3 does not include RSA key exchange, if server implementations use the same certificate for previous versions of TLS and TLS 1.3. Wagner and Schneier [41] developed a cross-cipher suite attack for SSLv3, in which an attacker could reuse a signed server key exchange message in a later exchange with a different cipher suite. Mavrogiannopoulos et al. [32] developed a cross-cipher suite attack allowing an attacker to use elliptic curve Diffie-Hellman as prime field Diffie-Hellman.

Attacks on export-grade cryptography. Recently, the FREAK [5] and Logjam [1] attacks allowed an active attacker to downgrade a connection to export-grade RSA and Diffie-Hellman, respectively. DROWN exploits export-grade symmetric ciphers, completing the export-grade cryptography attack trifecta.

9 Discussion

9.1 Implications for modern protocols

Although the protocol flaws in SSLv2 enabling DROWN are not present in recent TLS versions, many modern protocols meet a subset of the requirements to be vulnerable to a DROWN-style attack. For example:

1. RSA key exchange. TLS 1.2 [13] allows this.
2. Reuse of server-side nonce by the client. QUIC [10] allows this.
3. Server sends a message encrypted with the derived key before the client. QUIC, TLS 1.3 [37], and TLS False Start [30] do this.
4. Deterministic cipher parameters are generated from the premaster secret and nonces. This is the case for all TLS stream ciphers and TLS 1.0 block ciphers.

DROWN has a natural adaptation when all three properties are present. The attacker exposes a Bleichenbacher oracle by connecting to the server twice with the identical RSA ciphertexts and server-side nonces. If the RSA ciphertext is PKCS conformant, the server will respond with identical messages across both connections; otherwise they will differ.

9.2 Lessons for key reuse

DROWN illustrates the cryptographic principle that keys should be single use. Often, this principle is primarily applied to keys that are used to both sign and decrypt, but DROWN illustrates that using keys *for different protocol versions* can also be a serious security risk. Unfortunately, there is no widely supported way to pin X.509 certificates to specific protocols. While using per-protocol certificates may help defend against passive attacks, an active attacker could still leverage any certificate with a matching name.

9.3 Harms from obsolete cryptography

Recent years have seen a significant number of serious attacks exploiting outdated and obsolete cryptography. Many protocols and cryptographic primitives that were demonstrated to be weak decades ago are surprisingly common in real-world systems.

DROWN exploits a modification of an 18-year-old attack against a combination of protocols and ciphers that have long been superseded by better options: the SSLv2 protocol, export cipher suites, and PKCS #1 v1.5 RSA padding. In fact, support for RSA as a key exchange method, including the use of PKCS #1 v1.5, is mandatory even for TLS 1.2. The attack is made more severe by implementation flaws in rarely used code.

Our work serves as yet another reminder of the importance of removing deprecated technologies before they become exploitable vulnerabilities. In response to many of the vulnerabilities listed above, browser vendors have been aggressively warning end users when TLS connections are negotiated with unsafe cryptographic parameters, including SHA-1 certificates, small RSA and Diffie-Hellman parameters, and SSLv3 connections. This process is currently happening in a piecemeal fashion, primitive by primitive. Vendors and developers rightly prioritize usability and backward compatibility in standards, and are willing to sacrifice these only for practical attacks. This approach works less well for cryptographic vulnerabilities, where the first sign of a weakness, while far from being practically exploitable, can signal trouble in the future. Communication issues between academic researchers and vendors and developers have been voiced by many in the community, including Green [18] and Jager et al. [24].

The long-term solution is to proactively remove these obsolete technologies. There is movement towards this already: TLS 1.3 has entirely removed RSA key exchange and has restricted Diffie-Hellman key exchange to a few groups large enough to withstand cryptanalytic attacks long in the future. The CA/Browser forum will remove support for SHA-1 certificates this year. Resources such as the SSL Labs SSL Reports have gathered information about best practices and vulnerabilities in one place, in order to encourage administrators to make the best choices.

9.4 Harms from weakening cryptography

Export-grade cipher suites for TLS deliberately weakened three primitives to the point that they are now broken even to enthusiastic amateurs: 512-bit RSA key exchange, 512-bit Diffie-Hellman key exchange, and 40-bit symmetric encryption. All three deliberately weakened primitives have been cornerstones of high-profile attacks: FREAK exploits export RSA, Logjam exploits export Diffie-Hellman, and now DROWN exploits export symmetric encryption.

Like FREAK and Logjam, our results illustrate the continued harm that a legacy of deliberately weakened export-grade cryptography inflicts on the security of modern systems, even decades after the regulations influencing the original design were lifted. The attacks described in this paper are fully feasible against export cipher suites today. The technical debt induced by cryptographic "front doors" has left implementations vulnerable for decades. With the slow rate at which obsolete protocols and primitives fade away, we can expect some fraction of hosts to remain vulnerable for years to come.

Acknowledgements

The authors thank team Hashcat for making their GPUs available for the execution of the attack, Ralph Holz for providing early scan data, Adam Langley for insights about QUIC, Graham Steel for insights about TLS False Start, the OpenSSL team for their help with disclosure, Ivan Ristic for comments on session resumption in a BEAST-styled attack, and Tibor Jager and Christian Mainka for further helpful comments. We thank the exceptional sysadmins at the University of Michigan for their help and support throughout this project, including Chris Brenner, Kevin Cheek, Laura Fink, Dan Maletta, Jeff Richardson, Donald Welch, Don Winsor, and others from ITS, CAEN, and DCO.

This material is based upon work supported by the U.S. National Science Foundation under Grants No. CNS-1345254, CNS-1408734, CNS-1409505, CNS-1505799, CNS-1513671, and CNS-1518888, an AWS Research Education grant, a scholarship from the Israeli Ministry of Science, Technology and Space, a grant from the Blavatnik Interdisciplinary Cyber Research Center (ICRC) at Tel Aviv University, a gift from Cisco, and an Alfred P. Sloan Foundation research fellowship.

References

[1] ADRIAN, D., BHARGAVAN, K., DURUMERIC, Z., GAUDRY, P., GREEN, M., HALDERMAN, J. A., HENINGER, N., SPRINGALL, D., THOMÉ, E., VALENTA, L., VANDERSLOOT, B., WUSTROW, E., ZANELLA-BÉGUELIN, S., AND ZIMMERMANN, P. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security* (Oct. 2015).

[2] AL FARDAN, N. J., AND PATERSON, K. G. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 526–540.

[3] ALFARDAN, N. J., BERNSTEIN, D. J., PATERSON, K. G., POETTERING, B., AND SCHULDT, J. C. On the security of RC4 in TLS. In *22nd USENIX Security Symposium* (2013), pp. 305–320.

[4] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., SIMIONATO, L., STEEL, G., AND TSAY, J.-K. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 608–625.

[5] BEURDOUCHE, B., BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., KOHLWEISS, M., PIRONTI, A., STRUB, P.-Y., AND ZINZINDOHOUE, J. K. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy* (2015).

[6] BHARGAVAN, K., LAVAUD, A. D., FOURNET, C., PIRONTI, A., AND STRUB, P. Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 98–113.

[7] BHARGAVAN, K., AND LEURENT, G. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *Network and Distributed System Security Symposium* (Feb. 2016).

[8] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98*, vol. 1462 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998.

[9] BREYHA, W., DURVAUX, D., DUSSA, T., KAPLAN, L. A., MENDEL, F., MOCK, C., KOSCHUCH, M., KRIEGISCH, A., PÖSCHL, U., SABET, R., SAN, B., SCHLATTERBECK, R., SCHRECK, T., WÜRSTLEIN, A., ZAUNER, A., AND ZAWODSKY, P. Better crypto – applied crypto hardening, 2016. Available at https://bettercrypto.org/static/applied-crypto-hardening.pdf.

[10] CHANG, W.-T., AND LANGLEY, A. QUIC crypto, 2014. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit?pli=1.

[11] CVE-2015-0293. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0293.

[12] DE RUITER, J., AND POLL, E. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium* (Washington, D.C., Aug. 2015), USENIX Association.

[13] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878.

[14] DUONG, T., AND RIZZO, J. Here come the xor ninjas, 2011. http://netifera.com/research/beast/beast_DRAFT_0621.pdf.

[15] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., AND PAXSON, V. The matter of Heartbleed. In *14th Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 475–488.

[16] DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. ZMap: Fast Internet-wide scanning and its security applications. In *22nd USENIX Security Symposium* (Aug. 2013).

[17] FREIER, A., KARLTON, P., AND KOCHER, P. The secure sockets layer (SSL) protocol version 3.0. RFC 6101, 2011.

[18] GREEN, M. Secure protocols in a hostile world. In *CHES 2015* (Aug. 2015). https://isi.jhu.edu/~mgreen/CHESPDF.pdf.

[19] HAMILTON, R. QUIC discovery. https://docs.google.com/document/d/1i4m7DbrWGgXafHxwl8SwIusY2ELUe8WX258xt2LFxPM/edit#.

[20] Hashcat. http://hashcat.net.

[21] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium* (Aug. 2012).

[22] HICKMAN, K., AND ELGAMAL, T. The SSL protocol, 1995. https://tools.ietf.org/html/draft-hickman-netscape-ssl-00.

[23] HOLZ, R., AMANN, J., MEHANI, O., WACHS, M., AND KAAFAR, M. A. TLS in the wild: An Internet-wide analysis of TLS-based protocols for electronic communication. In *Network and Distributed System Security Symposium* (Geneva, Switzerland, Feb. 2016), S. Capkun, Ed., Internet Society.

[24] JAGER, T., PATERSON, K. G., AND SOMOROVSKY, J. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *Network and Distributed System Security Symposium* (2013).

[25] JAGER, T., SCHINZEL, S., AND SOMOROVSKY, J. Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. In *17th European Symposium on Research in Computer Security* (Berlin, Heidelberg, 2012), Springer Berlin Heidelberg, pp. 752–769.

[26] JAGER, T., SCHWENK, J., AND SOMOROVSKY, J. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In *22nd ACM Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1185–1196.

[27] KALISKI, B. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational), Mar. 1998. Obsoleted by RFC 2437.

[28] KÄSPER, E. Fix reachable assert in SSLv2 servers. OpenSSL patch, Mar. 2015. https://github.com/openssl/openssl/commit/86f8fb0e344d62454f8daf3e15236b2b59210756.

[29] KLIMA, V., POKORNÝ, O., AND ROSA, T. Attacking RSA-based sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems-CHES 2003*. Springer, 2003, pp. 426–440.

[30] LANGLEY, A., MODADUGU, N., AND MOELLER, B. Transport layer security (TLS) false start. *draft-bmoeller-tls-falsestart-00, June 2* (2010).

[31] LENSTRA, A. K., LENSTRA, H. W., AND LOVÁSZ, L. Factoring polynomials with rational coefficients. *Mathematische Annalen 261* (1982), 515–534. 10.1007/BF01457454.

[32] MAVROGIANNOPOULOS, N., VERCAUTEREN, F., VELICHKOV, V., AND PRENEEL, B. A cross-protocol attack on the TLS protocol. In *19th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 62–72.

[33] MEYER, C., AND SCHWENK, J. SoK: Lessons learned from SSL/TLS attacks. In *14th International Workshop on Information Security Applications* (Berlin, Heidelberg, Aug. 2013), WISA 2013, Springer-Verlag.

[34] MEYER, C., SOMOROVSKY, J., WEISS, E., SCHWENK, J., SCHINZEL, S., AND TEWS, E. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium*. USENIX Association, San Diego, CA, Aug. 2014, pp. 733–748.

[35] MÖLLER, B., DUONG, T., AND KOTOWICZ, K. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.

[36] OPENSSL. Change log. https://www.openssl.org/news/changelog.html#x0.

[37] RESCORLA, E., ET AL. The transport layer security (TLS) protocol version 1.3, draft.

[38] RIZZO, J., AND DUONG, T. The CRIME attack. EKOparty Security Conference, 2012.

[39] ROSKIND, J. QUIC design document, 2013. https://docs.google.com/a/chromium.org/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34.

[40] TURNER, S., AND POLK, T. Prohibiting secure sockets layer (SSL) version 2.0. RFC 6176 (Informational), Apr. 2011.

[41] WAGNER, D., AND SCHNEIER, B. Analysis of the SSL 3.0 protocol. In *2nd USENIX Workshop on Electronic Commerce* (1996).

[42] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *21st ACM Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 990–1003.

A Adaptations to Bleichenbacher's attack

A.1 Success probability of fractions

For a given fraction u/t, the success probability with a randomly chosen TLS conformant ciphertext can be computed as follows. Let m_0 be a random TLS conformant message, $m_1 = m_0 \cdot u/t$, and let ℓ_k be the expected length of the unpadded message. For $s = u/t \bmod N$ where u and t are coprime, m_1 will be SSLv2 conformant if the following conditions all hold:

1. m_0 is divisible by t. For a randomly generated m_0, this condition holds with probability $1/t$.
2. $m_1[1] = 0$ and $m_1[2] = 2$, or the integer $m \cdot u/t \in [2B, 3B)$. For a randomly generated m_0 divisible by t, this condition holds with probability
$$P = \begin{cases} 3 - 2 \cdot t/u & \text{for } 2/3 < u/t < 1 \\ 3 \cdot t/u - 2 & \text{for } 1 < u/t < 3/2 \\ 0 & \text{otherwise} \end{cases}$$
3. $\forall i \in [3, \ell_m - (\ell_k + 1)], m_1[i] \neq 0$, or all bytes between the first two bytes and the $(k+1)$ least significant bytes are non-zero. This condition holds with probability $(1 - 1/256)^{\ell_m - (\ell_k + 3)}$.
4. $m_1[\ell_m - \ell_k] = 0$: the $(\ell_k + 1)$st least significant byte is 0. This condition holds with probability $1/256$.

Using the above formulas for $u/t = 7/8$, the overall probability of success is $P = 1/8 \cdot 0.71 \cdot 0.37 \cdot 1/256 = 1/7{,}774$; thus the attacker expects to find an SSLv2 conformant ciphertext after testing 7,774 randomly chosen TLS conformant ciphertexts. The attacker can decrease the number of TLS conformant ciphertexts needed by multiplying each candidate ciphertext by several fractions.

Note that testing random s values until $c_1 = c_0 \cdot s^e \bmod N$ is SSLv2 conformant yields a success probability of $P_{rnd} \approx (1/256)^3 * (255/256)^{249} \approx 2^{-25}$.

A.2 Optimizing the chosen set of fractions

In order to deduce the validity of a single ciphertext, the attacker would have to perform a non-trivial brute-force search over all 5 byte `master_key` values. This translates into 2^{40} encryption operations.

The search space can be reduced by an additional optimization, relying on the fractional multipliers used in the first step. If the attacker uses $u/t = 8/7$ to compute a new SSLv2 conformant candidate, and m_0 is indeed divisible by $t = 7$, then the new candidate message $m_1 = m_0/t \cdot u$ is divisible by $u = 8$, and the last three bits of m_1 (and thus mk_{secret}) are zero. This allows reducing the searched `master_key` space by selecting specific fractions.

More generally, for an integer u, the largest power of 2 by which u is divisible is denoted by $v_2(u)$, and multiplying by a fraction u/t reduces the search space by a factor of $v_2(u)$. With this observation, the trade-off between the 3 metrics: the required number of intercepted ciphertexts, the required number of queries, and the required number of encryption attempts, becomes non-trivial to analyze.

Therefore, we have resorted to using simulations when evaluating the performance metrics for sets of fractions. The probability that multiplying a ciphertext by any fraction out of a given set of fractions results in an SSLv2 conformant message is difficult to compute, since the events are in fact inter-dependent: If $m \cdot 16/15$ is conforming, then m is divisible by 5, greatly increasing the probability that $m \cdot 4/5$ is also conforming. However, it is easy to perform a Monte Carlo simulation, where we randomly generate ciphertexts, and measure the probability that any fraction out of a given set produces a conforming message. The expected required number of intercepted ciphertexts is the inverse of that probability.

Formally, if we denote the set of fractions as F, and the event that a message m is conforming as $C(m)$, we perform a Monte Carlo estimation of the probability $P_F = P(\exists f \in F : C(m \cdot f))$, and the expected number of required intercepted ciphertexts equals $1/P_F$. The required number of oracle queries is simply $1/P_F \cdot |F|$. Accordingly, the required number of server connections is $2 \cdot 1/P_F \cdot |F|$, since each oracle query requires two server connections. And as for the required number of encryption attempts, if we denote this number when querying with a given fraction $f = u/t$ as E_f, then $E_f = E_{u/t} = 2^{40 - v_2(u)}$. We further define the required encryption attempts when testing a ciphertext with a given set of fraction F as $E_F = \sum_{f \in F} E_f$. Then the required number of encryption attempts in Phase 1 for a given set of fractions is $(1/P_F) \cdot E_F$.

We can now give precise figures for the expected number of required intercepted ciphertexts, connections to the targeted server, and encryption attempts. The results presented in Table 1 were obtained using the above approach with one billion random ciphertexts per fraction set F.

A.3 Rotation and multiplier speedups

For a randomly chosen s, the probability that the two most significant bytes are 0x00 02 is 2^{-16}; for a 2028-bit modulus N the probability that the next $\ell_m - \ell_k - 3$ bytes of m_2 are all nonzero is about 0.37 as in the previous section, and the probability that the $\ell_k + 1$ least significant delimiter byte is 0x00 is 1/256. Thus a randomly chosen s will work with probability $2^{-25.4}$ and the attacker expects to try $2^{25.4}$ values for s before succeeding.

However, since the attacker has already learned $\ell_k + 3$ most significant bytes of $m_1 \cdot R^{-1} \bmod N$, for $\ell_k \geq 4$ and $s < 2^{30}$ they do not need to query the oracle to learn if the two most significant bytes are SSLv2 conformant; they can compute this themselves from their knowledge of $\tilde{m_1} \cdot R^{-1}$. They iterate through values of s, test that the top two bytes of $\tilde{m_1} \cdot R^{-1} \bmod N$ are 0x00 02, and only query the oracle for s values that satisfy this test. Therefore, for a 2048-bit modulus they expect to test 2^{16} values offline per oracle query. The probability that a query is conformant is then $P = (1/256) * (255/256)^{249} \approx 1/678$, so they expect to perform 678 oracle queries before finding a fully SSLv2 conformant ciphertext $c_2 = (s \cdot R^{-1})^e c_1 \bmod N$.

We can speed up the brute force testing of 2^{16} values of s using algebraic lattices. We are searching for values of s satisfying $\tilde{m_1} R^{-1} s < 3B \bmod N$, or given an offset s_0 we would like to find solutions x and z to the equation $\tilde{m_1} R^{-1}(s_0 + x) = 2B + z \bmod N$ where $|x| < 2^{16}$ and $|z| < B$. Let $X = 2^{15}$. We can construct the lattice basis

$$L = \begin{bmatrix} -B & X\tilde{m_1}R^{-1} & \tilde{m_1}R^{-1}s_0 + B \\ 0 & XN & 0 \\ 0 & 0 & N \end{bmatrix}$$

We then run the LLL algorithm [31] on L to obtain a reduced lattice basis V containing vectors v_1, v_2, v_3. We then construct the linear equations $f_1(x,z) = v_{1,1}/B \cdot z + v_{1,2}/X \cdot x + v_{1,3} = 0$ and $f_2(x,z) = v_{2,1}/B \cdot z + v_{2,2}/X \cdot x + v_{2,3} = 0$ and solve the system of equations to find a candidate integer solution $x = \tilde{s}$. We then test $s = \tilde{s} + s_0$ as our candidate solution in this range.

$\det L = XZN^2$ and $\dim L = 3$, thus we expect the vectors v_i in V to have length approximately $|v_i| \approx (XZN^2)^{1/3}$. We will succeed if $|v_i| < N$, or in other words $XZ < N$. $N \approx 2^{8\ell_m}$, so we expect to find short enough vectors. This approach works well in practice and is significantly faster than iterating through 2^{16} possible values of $\tilde{s}$ for each query.

In summary, given an SSLv2 conformant ciphertext $c_1 = m_1^e \bmod N$, we can efficiently generate an SSLv2 conformant ciphertext $c_2 = m_2^e \bmod N$ where $m_2 = s \cdot m_1 \cdot R^{-1} \bmod N$ and we know several most significant bytes of m_2, using only a few hundred oracle queries in expectation. We can iterate this process as many times as we like to continue generating SSLv2 conformant ciphertexts c_i for which we know increasing numbers of most significant bytes, and which have a known multiplicative relationship to our original message c_0.

A.4 Rotations in the general DROWN attack

After the first phase, we have learned an SSLv2 conformant ciphertext c_1, and we wish to shift known plaintext bytes from least to most significant bits. Since we learn the least significant 6 bytes of plaintext of m_1 from a successful oracle $\mathcal{O}_{\text{SSLv2-export}}$ query, we could use a shift of 2^{-48} to transfer 48 bits of known plaintext to the most significant bits of a new ciphertext. However, we perform a slight optimization here, to reduce the number of encryption attempts. We instead use a shift of 2^{-40}, so that the least significant byte of $m_1 \cdot 2^{-40}$ and $\tilde{m_1} \cdot 2^{-40}$ will be known. This means that we can compute the least significant byte of $m_1 \cdot 2^{-40} \cdot s \bmod N$, so oracle queries now only require 2^{32} encryption attempts each. This brings the total expected number of encryption attempts for each shift to $2^{32} * 678 \approx 2^{41}$.

We perform two such plaintext shifts in order to obtain an SSLv2 conformant message, m_3 that resides in a narrow interval of length at most $2^{8\ell-66}$. We can then obtain a multiplier s_3 such that $m_3 \cdot s_3$ is also SSLv2 conformant. Since m_3 lies in an interval of length at most $2^{8\ell-66}$, with high probability for any $s_3 < 2^{30}$, $m_3 \cdot s_3$ lies in an interval of length at most $2^{8\ell_m - 36} < B$, so we know the two most significant bytes of $m_3 \cdot s_3$. Furthermore, we know the value of the 6 least significant bytes after multiplication. We therefore test possible values of s_3, and for values such that $m_3 \cdot s_3 \in [2B, 3B)$, and $(m_3 \cdot s_3)[\ell_m - 5] = 0$, we query the oracle with $c_3 \cdot s_3^e \bmod N$. The only condition for PKCS conformance which we haven't verified before querying the oracle is the requirement of non-zero padding, which holds with probability 0.37.

In summary, after roughly $1/0.37 = 2.72$ queries we expect a positive response from the oracle. Since we know the value of the 6 least significant bytes after multiplication, this phase does not require performing an exhaustive search. If the message is SSLv2 conformant after multiplication, we know the symmetric key, and can test whether it correctly decrypts the `ServerVerify` message.

A.5 Adapted Bleichenbacher iteration

After we have bootstrapped the attack using rotations, the original algorithm proposed by Bleichenbacher can be applied with minimal modifications.

The original step obtains a message that starts with the required 0x00 02 bytes once in roughly every two queries on average, and requires the number of queries to be roughly $16\ell_m$. Since we know the value of the 6 least significant bytes after multiplying by any integer, we can only query the oracle for multipliers that result in a zero 6th least significant byte, and again an exhaustive search over keys is not required. However, we cannot ensure

that the padding is non-zero when querying, which again holds with probability 0.37. Therefore, for a 2048-bit modulus, the overall expected number of queries for this phase is roughly $2048 * 2/0.37 = 11,070$.

A.6 Special DROWN MITM performance

For the first step, the probability that the three padding bytes are correct remains unchanged. The probability that all the intermediate padding bytes are non-zero is now slightly higher, $P_1 = (1-1/256)^{229} = 0.41$, yielding an overall maximal success probability $P = 0.1 \cdot 0.41 \cdot \frac{1}{256} = 1/6,244$ per oracle query. Since the attacker now only needs to connect to the server once per oracle query, the expected number of connections in this step is the same, 6,243. Phase 1 now yields a message with 3 known padding bytes and 24 known plaintext bytes.

For the remaining rotation steps, each rotation requires an expected 630 oracle queries. The attacker could now complete the original Bleichenbacher attack by performing 11,000 sequential queries in the final phase. However, with this more powerful oracle it is more efficient to apply a rotation 10 more times to recover the remaining plaintext bits. The number of queries required in this phase is now $10 \cdot 256/0.41 \approx 6,300$, and the queries for each of the 10 steps can be executed in parallel.

Using multiple queries per fraction. For the $\mathcal{O}_{\text{SSLv2-extra-clear}}$ oracle, the attacker can increase their chances of success by querying the server multiple times per ciphertext and fraction, using different cipher suites with different key lengths. They can negotiate DES and hope the 9th least significant byte is zero, then negotiate 128-bit RC4 and hope the 17th least significant byte is zero, then negotiate 3DES and hope the 25th least significant is zero. All three queries also require the intermediate padding bytes to be non-zero. This technique triples the success probability for a given pair of (ciphertext, fraction), at a cost of triple the queries. Its primary benefit is that fractions with smaller denominators (and thus higher probabilities of success) are now even more likely to succeed.

For a random ciphertext, when choosing 70 fractions, the probability of the first zero delimiter byte being in one of these three positions is 0.01. Hence, the attacker can use only 100 recorded ciphertexts, and expect to use $100 * 70 * 3 = 21,000$ oracle queries. For the Extra Clear oracle, each query requires one SSLv2 connection to the server. After obtaining the first positive response from the oracle, the attacker proceeds to phase 2 using 3DES.

A.7 Special DROWN with combined oracles

Using the Leaky Export oracle, the probability that a fraction u/t will result in a positive response is $P = P_0 * P_3$, where the formula for computing $P_0 = P((m \cdot u/t)[1,2] = 00||02)$ is provided in Appendix A.1, and P_3 is, for a 2048-bit modulus:

$$P_3 = P(\texttt{0x00} \notin \{m_3, \ldots, m_{10}\} \wedge \texttt{0x00} \in \{m_{11}, \ldots, m_\ell\}) \quad (1)$$
$$= (1-1/256)^8 * (1-(1-1/256)^{246}) = 0.60$$

Phase 1. Our goal for this phase is to obtain a divisor t as large as possible, such that $t|m$. We generate a list of fractions, sorted in descending order of the probability of resulting in a positive response from $\mathcal{O}_{\text{SSLv2-export-leaky}}$. For a given ciphertext c, we then query with the 50 fractions in the list with the highest probability, until we obtain a first positive response for a fraction u_0/t_0. We can now deduce that $t_0|m$. We then generate a list of fractions u/t where t is a multiple of t_0, sort them again by success probability, and again query with the 50 most probable fractions, until a positive answer is obtained, or the list is exhausted. If a positive answer is obtained, we iteratively re-apply this process, until the list is exhausted, resulting in a final fraction u^*/t^*.

Phase 2. We then query with all fractions denominated by t^*, and hope the ciphertext decrypts to a plaintext of one of seven possible lengths: $\{2,3,4,5,8,16,24\}$. Assuming that this is the case, we learn at least three least significant bytes, which allows us to use the shifting technique in order to continue the attack. Detecting plaintext lengths 8, 16 and 24 can be accomplished using three Extra Clear oracle queries, employing DES, 128-bit RC4 and 3DES, respectively, as the chosen cipher suite. Detecting plaintext lengths 2, 3, 4 and 5 can be accomplishing by using a single Leaky Export oracle query, which requires at most 2^{41} offline computation. In fact, the optimization over the key search space described in Section 3.2.1 is applicable here and can slightly reduce the required computation. Therefore, by initiating four SSLv2 connections and performing at most 2^{41} offline work, the attacker can test for ciphertexts which decrypt to one of these seven lengths.

In practice, choosing 50 fractions per iteration as described above results in a success probability of 0.066 for a single ciphertext. Hence, the expected number of required ciphertexts is merely $1/0.066 = 15$. The expected number of fractions per ciphertext for phase 1 is 60, as in most cases phase 1 consists of just a few successful iterations. Since each fraction requires a single query to $\mathcal{O}_{\text{SSLv2-export-leaky}}$, the overall number of queries for this stage is $15 * 60 = 900$, and the required offline computation is at most $900 * 2^{41} \approx 2^{51}$, which is similar to general DROWN. For a 2048-bit RSA modulus, the expected number of queries for phase 2 is 16. Each query consists of three queries to $\mathcal{O}_{\text{SSLv2-extra-clear}}$ and one query to $\mathcal{O}_{\text{SSLv2-export-leaky}}$, which requires at most 2^{41} computation. Therefore in expectancy the attacker has to perform 2^{45} offline computation for phase 2.