Last updated March 1, 2016. For contact information, test tools, and updates, visit **https://drownattack.com**.

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 that can decrypt passively collected TLS sessions from up-to-date clients by using a server supporting SSLv2 as a Bleichenbacher RSA padding oracle. We present two versions of the attack. The more general form exploits a combination of thus-far unnoticed protocol flaws in SSLv2 to develop a new and stronger variant of the Bleichenbacher attack. A typical scenario requires the attacker to observe 1,000 TLS handshakes, then initiate 40,000 SSLv2 connections and perform 2^{50} offline work to decrypt a 2048-bit RSA TLS ciphertext. (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 using Amazon EC2, at a cost of $440. 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. 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^{25} SSLv2 connections and 2^{65} offline work.

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

1 Introduction

TLS [14] 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 [35]. Prominent recent examples include attacks on outdated or deliberately weakened encryption in RC4 [3], RSA [6], and Diffie-Hellman [1], different side channels including Lucky13 [2], BEAST [15], and POODLE [37], and several attacks on invalid TLS protocol flows [6, 13, 7].

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" [46]. Most modern TLS clients do not support SSLv2 at all. However, in Internet-wide scans we found that out of 36 million HTTPS servers, 6 million (17%) support SSLv2.

Bleichenbacher's padding oracle attack [9] is an adaptive chosen ciphertext attack against RSA PKCS#1 v1.5, the RSA padding standard used in TLS. This attack enables decryption of RSA-encrypted 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 RSA decryption queries needed to deduce a plaintext. All widely-used modern SSL/TLS server implementations include countermeasures against Bleichenbacher attacks.

A Bleichenbacher attack on SSLv2. 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 and much more dangerous 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 named our 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 for 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 a heavily 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 the Amazon EC2 service.

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

Our *special* version of the DROWN attack, which exploits a flaw in OpenSSL for a more efficient oracle, requires roughly the same number of captured TLS sessions, half as many connections to the victim server, and no large computations. The resulting attack can be completed on a single core on commodity hardware in less than a minute, without GPUs or distributed computing, and is limited primarily by how fast the server can complete handshakes. It is fast enough to perform man-in-the-middle attacks on live TLS sessions before the handshake times out, even allowing the attacker to target connections to servers that prefer non-RSA cipher suites and *downgrade* a modern TLS client to RSA key exchange. Our Internet-wide scans suggest that 79% of HTTPS servers that are vulnerable to the general attack, namely 26% of all HTTPS servers, are also vulnerable to real-time attacks exploiting this dangerous implementation flaw.

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.

Responsible disclosure. The DROWN attack was assigned CVE-2016-0800. We disclosed our attacks to OpenSSL and worked with them to coordinate disclosure. The specific OpenSSL vulnerabilities we discovered have been assigned CVE-2015-3197 and CVE-2016-0703. In response to our disclosure, 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 their efforts to entirely remove support for the protocol from the NSS codebase. In response to our disclosure, Google will disable QUIC support for non-whitelisted servers, and make changes to the QUIC standard, as detailed in Section 7. We also notified IBM, Cisco, Amazon, the German CERT-Bund, and the Israeli CERT.

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 ($|N| = \ell$) 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 there are newer versions of the PKCS standard, for example RSA PKCS#1 v2.0 which implements OAEP, SSL/TLS uses PKCS#1 v1.5. The basic task of the PKCS#1 v1.5 encryption padding scheme [27] is to randomize encryptions by prepending a random padding string PS to a message k (typically a symmetric session key) before applying RSA encryption:

1. The plaintext message is k. The encrypter generates a random byte string PS, where $|PS| \geq 8$, $|PS| = \ell - 3 - |k|$, and `0x00` $\notin \{PS[1], \dots, PS[|PS|]\}$.
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 $m[1]||m[2]||...||m[\ell]$ 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 value $i > 10$ such that $m[i] = 0x00$. Then, it extracts $k = m[i+1]||\ldots||m[\ell]$. 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 [14]. 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 [45, 46] which were fixed in SSLv3 [19]. 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 much 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`).

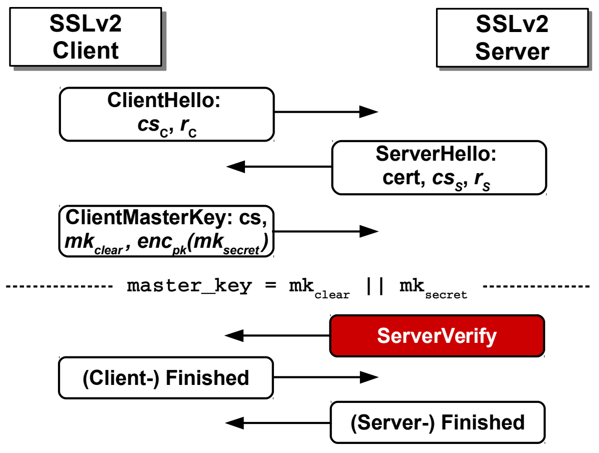
SSLv2 Client
SSLv2 Server
ClientHello: cs_c, r_c
ServerHello: cert, cs_s, r_s
ClientMasterKey: cs, mk_{clear}, $enc_{pk}(mk_{secret})$
master_key = mk_clear || mk_secret
ServerVerify
(Client-) Finished
(Server-) Finished

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.

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 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 [14] 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 OpenSSL SSLv2 cipher suite selection bug

The SSLv2 protocol is supported in OpenSSL by default in all versions under 1.1.0. OpenSSL removed SSLv2 cipher suites from the default cipher string in 2010 between versions 0.9.8n and 1.0.0; the changelog discusses this as being equivalent to disabling support for SSLv2 by default [38]. Unfortunately, during our experiments we discovered that OpenSSL servers do not respect the cipher suites advertised in the `ServerHello` message. That is, the client can select an *arbitrary* cipher suite in the `ClientMasterKey` message and force the use of export cipher suites even if they are explicitly disabled in the server configuration. The SSLv2 protocol itself was still enabled by default in the OpenSSL standalone server for the most recent OpenSSL versions prior to our disclosure.

We notified the OpenSSL team of this vulnerability, which was assigned CVE ID CVE-2015-3197. We have cooperated to develop a fix, which was included in OpenSSL releases 1.0.2f and 1.0.1r [38].

2.4 Bleichenbacher's attack

Bleichenbacher's attack is a padding oracle attack—it exploits the fact that RSA ciphertexts should decrypt to plaintexts compliant with the PKCS#1 v1.5 padding format. 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 [9] 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 he wishes to decrypt to discover the message m_0. He has no access to the private RSA key, but instead has 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 $B = 2^{8(\ell-2)}$. 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 [9].

Countermeasures. In order to protect against this attack, the decrypter must not leak any information about the PKCS#1 v1.5 validity of the ciphertext. Since the ciphertext itself does not decrypt to a valid message, the decrypter needs to generate a fake plaintext and continue with the protocol using 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 and finishes the handshake with this random premaster secret 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. 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 very powerful oracle, and describe our efficient special DROWN attack.

3.1 Attack scenario

We consider a server that accepts 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 where the private key is not known to the attacker.

Server RSA key exposed via SSLv2. 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.

The attacker's position in the network. Our attacker is able to passively eavesdrop on traffic between the client and server and record RSA-based TLS traffic, but does not perform any active man-in-the-middle interference.

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. If the attacker's intended victim is the *server*, rather than a specific client, observing this many connections from many clients might take only a short time for an attacker who is located at a company firewall or who could perform a DNS spoofing or BGP hijacking attack to redirect traffic transparently through themselves. If the attacker's intended victim is a *particular client*, this is still feasible in many cases. As an example, the Mozilla Thunderbird email client will check for new email messages every ten minutes by default. A targeted user will make 1,000 connections after leaving the application running for a week. A less patient attacker could embed or inject malicious JavaScript on an otherwise innocuous web site to cause the client to connect repeatedly to the victim server in a short time frame, as in the BEAST attack [15]. Normally such connections would use TLS session resumption instead of completing a fresh handshake on each time, but if an attacker can trigger an error, the next connection will be negotiated with a fresh handshake.

3.2 A generic SSLv2 oracle

Our attacks make use of a padding oracle that can be queried on a ciphertext and leaks information about decrypted plaintext; this abstractly models the information gained from an SSLv2 server's behavior. Our SSLv2 oracles reveal many bytes of plaintext, resulting in 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 structure of 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 k. In the following, we denote such a ciphertext to be *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 $k+3$ bytes of information about $m = c^d \bmod N$: the first two bytes are $00||02$, and the last $k+1$ bytes are $00||mk_{secret}$. The length k of mk_{secret} varies based on the cipher suite used in the instantiation of 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 information about m. For `SSL_DES_192_EDE3_CBC_WITH_MD5`, k is 24 bytes and the attacker learns 27 bytes of plaintext.

3.3 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 ciphertext transmitted in a TLS key exchange 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 needs to 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. [5], using such an oracle for Bleichenbacher's attack would require 12 million oracle queries.[1]

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

0. The attacker collects many encrypted TLS RSA key exchange messages.

1. He then attempts to convert the intercepted TLS ciphertexts containing a 48-byte premaster secret to valid RSA PKCS#1 v1.5 encoded ciphertexts containing messages of length appropriate to the SSLv2 oracle $\mathcal{O}$. We accomplish this by taking advantage of RSA ciphertext malleability and a technique of Bardou et al. [5].

[1] See Table 1 in [5]. The oracle is denoted with the term `FFF`.

2. Once the attacker has obtained a valid SSLv2 RSA ciphertext, he can continue with a modified version of Bleichenbacher's attack, and decrypt the message after many more oracle queries.

3. The attacker can then transform 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.

3.3.1 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. A trivial approach would be to generate multipliers $s_i \in \{s_1, s_2, \ldots\}$, and compute ciphertexts $c_i = (c_0 s_i{}^e) \bmod N$, until one gets accepted by $\mathcal{O}$. However, the number of generated ciphertexts would be high, because $\mathcal{O}$ is very restrictive; for 2048-bit RSA keys and an oracle returning a 5-byte k the probability that a random ciphertext becomes SSLv2 conformant is $P_{rnd} \approx (1/256)^3 * (255/256)^{249} \approx 2^{-25}$.

Instead, we rely on the concept of *trimmers*, which were introduced by Bardou et al. [5]. 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 with a fraction u/t, the resulting number will lie near the original message: $m_0 \approx m_0/t \cdot u$. We shall refer to such fractions as "small" fractions.

This method allows us to generate new SSLv2 conformant messages with a much higher probability. Let c_0 be an intercepted TLS conformant RSA ciphertext, and let $m_0 = c_0^d \bmod N$ be its corresponding 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 $k = 5$, and consider the fraction $u = 7, t = 8$. The probability that a random ciphertext c_0 will be SSLv2 conformant is 1/7,774, so we expect to make 7,774 oracle queries before discovering a ciphertext c_0 for which $c_0 u/t$ is SSLv2 conformant, much better than a randomly selected multiplier. Appendix B.1 gives more details on computing these probabilities.

3.3.2 Shifting known plaintext bytes

Once we have obtained an SSLv2 conformant ciphertext c_1, we have also learned from our oracle information about the $k+1$ 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 bits 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 $B = 2^{8(\ell-2)}$. Abusing notation slightly, let the integer $m_1 = 2 \cdot B + PS \cdot R + k$ be the plaintext satisfying $m_1^e = c_1 \bmod N$. At this stage, the k-byte integer mk_{secret} is known and the $\ell - k - 3$-byte integer PS is not.

Let $\tilde{m_1} = 2 \cdot B + k$ 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 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, but we know that it consists of $\ell - k - 3$ bytes. This means that the known value $\tilde{m_1} \cdot R^{-1}$ shares most of its $k+3$ 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 advantage of 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 B.3 gives more details.

3.3.3 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 to continue using Bleichenbacher's original attack, once we have used the above techniques to obtain a SSLv2 conformant message m_3 and an integer s_3 such that $m_3 \cdot s_3$ is SSLv2 conformant. At this point, we can apply the original algorithm proposed by Bleichenbacher as described in Section 2.4, with minimal modifications.

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 $k+1$ least significant bytes after multiplying by any integer, we can query the oracle only on multipliers that cause the $(k+1)$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 any correct SSLv2 implementation that accepts export-grade cipher suites can be used as a padding oracle. We then show how to adapt the techniques described in Section 3.3 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 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 sequence of 11 bytes as the clear portion of the `master_key`, 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}. He then computes the corresponding `server_write_key` and checks whether the `ServerVerify` message decrypts to the `challenge`. One value should pass this check; let this value be termed mk_0. Recall that if the RSA plaintext was valid, mk_0 is the unpadded data in the RSA plaintext. 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 ciphertext was valid, the attacker can directly decrypt a correct `challenge` value from the `ServerVerify` message by using the `master_key` mk_0. Otherwise, if the `ServerVerify` message does not correctly decrypt to the `challenge`, the RSA ciphertext was invalid, and the attacker knows the mk_0 value was generated at random.

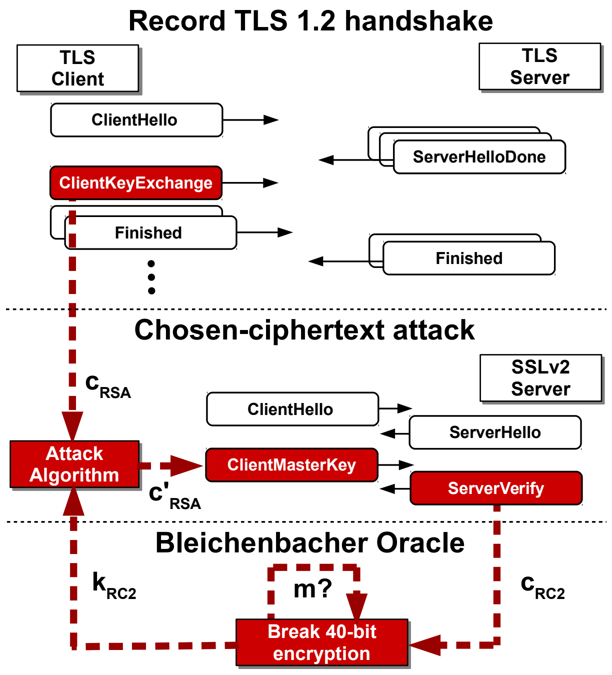

Figure 2: **Our SSLv2-based Bleichenbacher attack on TLS.** An attacker passively collects RSA ciphertexts from a TLS 1.2 handshake, and then performs oracle queries against a server that supports SSLv2 with the same public key to decrypt the TLS ciphertext.

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.

If the server does not support 40-bit export ciphers, the attack can also be mounted in feasible computation time by choosing DES as the symmetric cipher. Choosing DES means the exhaustive search is now done over a key space of 56 bits, thus increasing the cost of the attack by a factor of 2^{16}, but does not fundamentally change anything except the increased cost.

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.

4.2.1 Attack scenario

As described in Section 3.1, 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.

Server supports export cipher suites for SSLv2. We also assume the server supports export cipher suites for SSLv2. This can happen for two reasons. First, the same servers that fail to follow best practices in disabling SSLv2 [45] may also fail to follow best practices by supporting export cipher suites. Alternatively, the servers might be running a version of OpenSSL prior to January 2016, in which case they are vulnerable to the OpenSSL cipher suite selection bug described in Section 2.3, and an attacker may negotiate a cipher suite of his choice independent of the server configuration.

Correct Bleichenbacher countermeasure. We assume the server implements the recommended countermeasure against Bleichenbacher's attack in all protocol versions, including SSLv2. If the decrypted RSA ciphertext has invalid padding, the server generates a random premaster secret or `master_key` and continues the handshake with this random string. We assume this countermeasure is implemented correctly and the server is neither vulnerable to timing nor flush-and-reload side-channel attacks [36, 49].

Computing power. 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.

4.2.2 Constructing the attack

The attacker can exploit the SSLv2 vulnerability as illustrated in Figure 2, following the generic attack outline described in Section 3.3 and has several distinct phases:

0. He passively collects 1,000 TLS handshakes from connections using RSA key exchange.
1. The attacker then attempts 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 3.3.1, and querying $\mathcal{O}_{\mathsf{SSLv2\text{-}export}}$. 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 m_1, he uses the shifting technique explained in Section 3.3.2 to find an integer s_1 such that $m_2 = m_1 \cdot 2^{-40} \cdot s_1$ is also SSLv2 conformant. Appendix B.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. He then searches 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 3.3.3, 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 B.5.

4.2.3 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 3.3.1.

Table 1 highlights a few choices for F and the resulting performance metrics for 2048-bit RSA keys. Appendix B.6 provides more details on the derivation of these numbers and other possible optimization choices. Table 2 gives the expected number of Bleichenbacher queries for different RSA key sizes, when minimizing total oracle queries.

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.

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 measures the calculation of an MD5 hash and the RC2 decryption. Our optimized implementation gave a final speed of 515MH/s, a speedup factor of 19.8.

We obtained our improvements through a number of optimizations. 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 optimized the cryptographic checks themselves by rewriting the RC2 implementation to use 32-bit instructions, removing unnecessary RC2 keysize checks, dropping unused ADD instructions during MD5, and manually shifting input bytes into the MD5 input registers to avoid loop branches. We describe these optimizations in further detail in Appendix C.

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[21] 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) [4] service. We used a cluster composed of 200 variable-price "spot" instances: 150 `g2.2xlarge` instances, each of which contains 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. See Appendix D for more details.

5 Special DROWN

We discovered a vulnerability in recent (but not current) versions of the OpenSSL SSLv2 handshake code that creates a powerful Bleichenbacher oracle, and drastically reduces the amount of computation required to implement our attack. The vulnerability, which has been designated CVE-2016-0703, was present in the OpenSSL codebase from at least the start of the repository, in 1998, until it was unknowingly fixed on March 4, 2015 by a patch [28] designed to correct an unrelated problem [12]. By adapting DROWN to exploit this special case, we can cut the number of connections required by more than 50% and reduce the computational work to a negligible amount.

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 the first byte, the attacker makes another connection with the same RSA ciphertext but 14 bytes of clear key, resulting in the `master_key`:

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

Since the attacker already knows $k[1]$, he can easily brute force the second byte. 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.

This session key-recovery attack can be directly converted to a Bleichenbacher oracle. Given a candidate ciphertext and symmetric key length k, the attacker sends the ciphertext with 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 k known bytes.
- If the ciphertext is invalid, the `master_key` will be replaced with 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 k-byte encrypted key, they can recover the 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, and thus the two most significant bytes 0x00 02 of the plaintext m. After k connections, the attacker can additionally learn the 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 TLS decryption with special DROWN

Using our oracle $\mathcal{O}_{\text{SSLv2-extra-clear}}$, we can construct an extremely efficient version of our TLS decryption attack. The OpenSSL extra clear oracle provides three significant advantages over our export oracle $\mathcal{O}_{\text{SSLv2-export}}$: (1) It no longer requires an export cipher suite, and, in fact, we gain efficiency by exploiting regular SSLv2 ciphers; (2) It requires only one handshake per oracle query; and (3) Computation is reduced to one `ServerVerify` decryption per oracle query, versus 2^{40}.

5.2.1 Attack scenario

As before, we consider a server that accepts TLS connections, and a client that negotiates a secure, state-of-the-art TLS version with a `TLS_RSA` cipher suite. The same RSA key pair used for TLS is also used on a server that is running a vulnerable version of OpenSSL.

5.2.2 Constructing the attack

The attacker can exploit the OpenSSL extra clear vulnerability to efficiently decrypt a TLS ciphertext as follows. We will use the cipher suite `SSL_DES_192_EDE3_CBC_WITH_MD5` as the cipher suite, allowing the attacker to recover 24 bytes of key at a time from the oracle. We first present a straightforward adaptation of the general DROWN attack to the extra clear oracle, before later applying a few additional optimizations made possible by this new oracle.

0. The attacker intercepts several hundred TLS handshakes using RSA key exchange.
1. The attacker uses the fractional trimmers as described in Section 3.3.1 to convert the TLS ciphertexts into an SSLv2 conformant ciphertext c_0.
2. Once the attacker has obtained a valid SSLv2 ciphertext c_1, he repeatedly uses the shifting technique described in Section 3.3.2 to rotate the message by 25 bytes each iteration, learning 27 bytes with each shift. After several iterations, he has learned the entire plaintext.
3. The attacker then transforms the decrypted SSLv2 plaintext into the decrypted TLS plaintext.

Attack costs Using 40 fractional trimmers, this more efficient oracle attack allows the attacker to recover one in 260 TLS session keys using only about 17,000 connections to the server. The computation cost is so low that we can complete the full attack on a single workstation in under one minute. Appendix B.7 gives more details.

Mounting the attack using the optimized version of Special DROWN described in Appendix B.7 allows the attacker to target one of 100 connections, at the expense of increasing the number of queries to 27,000.

5.3 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

		All Certificates			*Trusted certificates*		
Protocol	**Port**	**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 1M)	443	611 K	82 K (13%)	152 K (25%)	456 K	38 K (8%)	109 K (24%)

Table 3: **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.

connections between modern browsers and TLS servers—even those configured to prefer non-RSA cipher suites.

Attack scenario. 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].

Using the fully optimized version of special DROWN, the attack still requires intercepting an average of 100 ciphertexts, only one of which will be decrypted, 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 3.1. 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.

Once the attacker has obtained a positive response, he can proceed to the final phase of the special DROWN attack described above, which employs 200-bit rotation 10 times to fully decrypt the plaintext. Our current implementation requires under 30 seconds for this phase on a single PC.

The ability of the victim server to perform 17,000 handshakes in less than a minute is not an impediment for modern hardware. An RSA private key operation with a 2048-bit modulus requires on the order of 1 ms using OpenSSL on a recent-generation CPU, so the cryptographic portion of the attacker's queries induces additional server load of roughly 14 core-seconds. In tests with a nearby server running Apache 2.4, we could easily complete 10,000 HTTPS requests in under 10 seconds.

6 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. 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 [17] 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 3. 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

		Any certificate			*Trusted certificates*		
Protocol	**Port**	**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 1M)	443	611 K	22 K (4%)	52 K (9%)	456 K (100%)	33 K (7%)	85 K (19%)

Table 4: **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.

versions and cipher suites, under the assumption that even bad or obsolete encryption is better than plaintext [10]. 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.

Widespread public key reuse. Reuse of RSA key material across hosts and certificates is widespread, as has been documented in [23, 34]. In many cases this is benign: many organizations issue multiple TLS certificates for distinct domains (e.g. one for each TLD) with the same public key; reusing the same key simplifies the use of SSL 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 [16].

There are many reasons why the same public key or certificate would be reused across different ports and services within an organization. For example a mail server that serves SMTP, POP3, and IMAP from the same daemon would likely share the same TLS configuration. Additionally, an organization might choose to purchase a single wildcard TLS certificate, and use it on both web servers and mail servers. Public keys have also been observed to be widely shared across independent organizations due to default certificates and public keys that are shipped with networked devices and software, improperly configured virtual machine images, and random number generation flaws.

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 the number of vulnerable hosts rises to 33% when considering RSA keys used by another service that is vulnerable to DROWN. Appendix A gives more detailed statistics on the reuse of RSA key material across hosts and ports.

Special DROWN. As shown in Table 4, 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 there are 2.7 M public keys that are vulnerable to general DROWN, we find 1.1 M vulnerable to special DROWN (41% as many). Vulnerability among Alexa Top Million domains is lower, with only 9% of Alexa 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 a key exposed by a special DROWN oracle is vulnerable to a impersonation attacks. Extending our search to account for shared names, we find 3.8 M (22%) of hosts with browser-trusted certificates are vulnerable to man-in-the-middle, as well as 19% of the browser-trusted Alexa Top Million.

7 Signature forgery attacks and QUIC

An attacker can also use a Bleichenbacher-type attack to compute valid RSA signatures on arbitrary messages. Mathematically, RSA signing and decryption are identical. Such an attack could theoretically be used to forge a signed Server Key Exchange message for Diffie-Hellman cipher suites, thus allowing an attacker to perform a man-in-the-middle attack against all TLS versions up to TLSv1.3. [26] Since the server key exchange message includes the client and server randoms, the attacker must forge the signature online before the handshake times out. We are not able to use all of our optimizations for signature forgery, so such an attack does not seem feasible without additional improvements, even for special DROWN.

7.1 Extending the attack to QUIC

However, our attack can be extended to a feasible-time man-in-the-middle attack against QUIC [26]. QUIC [42, 11] 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 successfully mount such an attack against a server who exposed their RSA public keys via SSLv2.

A QUIC client receives a "server config" message enumerating connection parameters, a static elliptic curve Diffie-Hellman public value, and a validity period that is signed by the server's public key. An attacker could generate a Diffie-Hellman public value for which he knows the private key, and set the expiration date far in the future in order to mount a man-in-the-middle attack against any client.

Unauthenticated QUIC discovery. In order to mount the attack, the attacker needs to present a forged QUIC config to the client. This is straightforward, since QUIC discovery may happen over non-encrypted HTTP [20]. The server does not even need to support QUIC at all: an attacker could impersonate the attacked server over an unencrypted 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]

Signature forgery details. The attack proceeds much as in Section 3.3, except that we are not able to use some of the optimizations so it is more expensive.

The first step is to discover a valid, PKCS conformant SSLv2 ciphertext. In the case of TLS decryption, our input ciphertext was PKCS conformant to begin with; this is not the case for our QUIC message c_0. Thus for the first phase, we iterate through possible multiplier values s until the attacker randomly encounters a valid SSLv2 message in $c_0 \cdot s$. For 2048-bit RSA keys, the probability of this random event is $P_{rnd} \approx 2^{-25}$; see Section 3.3 for the computation.

Once the first SSLv2 conformant message is found, the attacker proceeds with the signature forgery as he would in Step 2 of the attack against TLS. 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 $2^{25} = 34$ million expected queries in the first phase, above. At a rate of 388 queries/second, an attacker would finish in one day; at a rate of 12 queries/second an attacker 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 dollars and could be parallelized to Amazon's capacity.

For the extra clear oracle, there is only negligible computation per oracle query, so the computational cost for the first phase is 2^{25}.

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.

7.2 SSLv2 servers with CA certificates

Some web servers support SSLv2 while presenting a CA certificate, which can be used to issue further leaf certificates. In that case, an attacker could create his own certificate and use the vulnerable server to forge a CA signature over his certificate by executing an attack similar to the above. The number of queries is identical to the number of queries required for the attack against QUIC. This attack would allow the attacker to impersonate any website against any client trusting the CA certificate.

We did not observe any trusted CA certificates used on vulnerable servers. We did, however, observe a number of routers that supported SSLv2 while presenting CA certificates that are untrusted by modern browsers.

8 Related work

Bleichenbacher's attack. Bleichenbacher's adaptive chosen ciphertext attack against SSL was first published in 1998 [9]. Since then, several works have adapted his attack to different scenarios [29, 5, 25].

Despite the fact that the TLS standard [14] explicitly introduces countermeasures against Bleichenbacher's attack, several modern implementations have been discovered to be vulnerable to it in recent years. Meyer *et al.* [36] inspected various software and hardware implementations and discovered timing side-channels that enabled the attack. Zhang *et al.* applied Bleichenbacher's attack to develop a cache flush-and-reload timing attack against OpenSSL in cross-tenant environments [49]. These side-channel attacks, however, are applicable only in scenarios where the attacker is physically close to or co-located with the victim and are based on implementation failures.

Cross-protocol attacks. Jager et al. [26] observed 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 [46] 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 [33] developed a cross-cipher suite attack allowing an attacker to use elliptic curve Diffie-Hellman as plain Diffie-Hellman.

Attacks on export-grade cryptography. Recently, the FREAK [6] and Logjam [1] attacks allowed an active attacker to downgrade a connection to export-grade RSA and Diffie-Hellman, respectively. Export-grade cryptography plays an important role in DROWN as well, as it exploits export-grade symmetric ciphers.

Further attacks on SSL/TLS. Other attacks on SSL and TLS include: POODLE [37], which exploits SSLv3's lack of a requirement for the contents of padding bytes, and its MAC-then-encrypt construction; CRIME [41], which exploits support for compression and observes ciphertexts' lengths in order to decrypt traffic; The RC4 Biases attack [3], which utilizes biases in the the RC4 keystream; Lucky13 [2], which exploits small timing differences and MAC-then-encrypt; and BEAST [15], which exploits predictable IVs in TLS. Bhargavan and Leurent presented SLOTH attacks and broke TLS and other protocols using MD5 for computing transcript hashes [8].

9 Discussion

9.1 Lessons for protocol design

A natural question is to ask whether SSLv3 or later versions of TLS could also be vulnerable. Our attack exploits two properties of the SSLv2 protocol:

Server authenticates first. First, the fact that in SSLv2 the server responds to the `ClientMasterKey` message before the client proves it has knowledge of the RSA plaintext, provides a direct message side channel. In SSLv3 and later, the client must demonstrate knowledge of the RSA plaintext first via a valid `ClientFinished` message before the server sends a message derived from the RSA plaintext. In order to perform a similar attack in this case, the client would need to perform an online brute-force attack.

Short secrets. Second, SSLv2 allows RSA plaintexts that are short enough to be vulnerable to a feasible-time brute force search. For export ciphers, the unpadded RSA plaintext is five bytes long. In SSLv3 and later versions of TLS, the RSA plaintexts and premaster secret length is 48 bytes, even for export ciphers with 40-bit strength. For later protocol versions, an attacker can perform a brute-force search over the derived 40-bit key if a client negotiates an export cipher suite, but the 48-byte premaster secret length appears to prevent an attacker from escalating the weakness of the export cipher strength into a similar protocol vulnerability.

9.2 Implications for modern protocols

Modern TLS versions are not vulnerable to the precise attack given in this paper, but they have similar properties that might allow a related attack.

Although we do not present concrete attacks on modern protocols, we argue that modern practices of cryptographic protocol design do not include a systematic analysis to prevent direct message side channel Bleichenbacher attacks. A hypothesized protocol with modern parameters would be vulnerable to such an attack if it has the following properties:

1. RSA key exchange. TLS 1.2 [14] allows this.
2. It allows re-use of server-side nonce by the client. QUIC [11] allows this.
3. The server sends the first message encrypted using a key derived from the asymmetric key exchange. QUIC, TLS 1.3 [39], and TLS False Start [30] exhibit this property.

When all three properties are combined, a natural adaptation of our attack presents itself. The attacker obtains a Bleichenbacher oracle by connecting to the server twice with the same RSA ciphertext and the same server-side nonce, and comparing the messages sent by the server. If the RSA ciphertext is PKCS conformant, the two messages will be identical. Otherwise, they will differ. Note that we also assumed that all symmetric cipher parameters, including IVs for block ciphers, are deterministically generated from the premaster secret and nonces; this is the case for TLS 1.0. If that is not the case, in most realistic configurations, the attacker can choose a stream cipher.

An attacker can use False Start to cause a victim client to perform TLS handshakes using RSA for key exchange, even if the server supports other key exchange methods which provide Perfect Forward Secrecy. The attacker masquerades as the server and indicates support for RSA key exchange only. The client will then handshake using

RSA, and send application layer data, before the server authenticates by sending the `Finished` message. The False Start standard indeed discourages the use of RSA as the key exchange method, but does not explicitly forbid it, leaving the security of the protocol dependent on correct choices in the client configuration. Our attacks show that relying on such assumptions is extremely brittle protocol design.

9.3 Lessons for key reuse

Our attacks also illustrate another important cryptographic principle: that keys should be single use. For public keys we think of this principle as applying primarily to keys that are used to both sign and decrypt, but our attacks illustrate that using keys *for different protocol versions* can also be a serious security risk. Unfortunately, the TLS certificate authority funding model produces a financial incentive for users to purchase as few certificates as necessary to protect their infrastructure. However, even without this financial incentive in place, the sheer number of SSL/TLS protocol versions in use would make key management difficult.

9.4 Harms from obsolete cryptography

Recent years have seen a significant number of serious attacks exploiting outdated and obsolete cryptography. Many of these protocols and cryptographic primitives are surprisingly common in deployed systems even decades after they were demonstrated to be weak.

The attack described in this paper 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, and are willing to sacrifice these only for practical attacks. This standard 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 [32] and Jager et al. [24].

The long-term solution is to proactively remove these obsolete technologies. There has been a movement towards this already: TLS 1.3 has removed RSA key exchange entirely 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. And 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.

9.5 Harms from deliberately weakening cryptography

Export-grade cipher suites for TLS deliberately weakened three primitives to the point that they are broken even to enthusiastic amateurs today: 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 attack against export RSA, Logjam against Diffie-Hellman, and our DROWN attack against export-grade symmetric cryptography.

Our results illustrate, like FREAK and Logjam, 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; against even DES they would be at the limits of the computational power available to an attacker. The technical debt induced by cryptographic "front doors" has left implementations vulnerable for decades. Together with the slow rate at which obsolete protocols and primitives entirely disappear, we can expect some fraction of hosts to continue to be 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.

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. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. National Science Foundation.

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] Amazon EC2. https://aws.amazon.com/ec2/.

[5] 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.

[6] 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).

[7] 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.

[8] Bhargavan, K., and Leurent, G. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *NDSS* (Feb. 2016).

[9] 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.

[10] 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.

[11] Chang, W.-T., and Langley, A. QUIC crypto, 2014. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit?pli=1.

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

[13] de Ruiter, J., and Poll, E. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium* (Washington, D.C., Aug. 2015), USENIX Association.

[14] 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.

[15] Duong, T., and Rizzo, J. Here come the xor ninjas. *Unpublished manuscript* (2011), 4.

[16] 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 *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 475–488.

[17] Durumeric, Z., Wustrow, E., and Halderman, J. A. ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security Symposium* (Aug. 2013).

[18] Fluhrer, S., Mantin, I., and Shamir, A. *Selected Areas in Cryptography: 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16–17, 2001 Revised Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, ch. Weaknesses in the Key Scheduling Algorithm of RC4, pp. 1–24.

[19] Freier, A., Karlton, P., and Kocher, P. The secure sockets layer (SSL) protocol version 3.0. RFC 6101, 2011.

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

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

[22] Hickman, K., and Elgamal, T. The SSL protocol, 1995. Available at https://tools.ietf.org/html/draft-hickman-netscape-ssl-00.

[23] Holz, R., Amann, J., Mehani, O., Wachs, M., and Kâafar, M. A. TLS in the wild: an Internet-wide analysis of TLS-based protocols for electronic communication. *CoRR abs/1511.00341* (2015).

[24] Jager, T., Paterson, K. G., and Somorovsky, J. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *NDSS* (2013).

[25] Jager, T., Schinzel, S., and Somorovsky, J. *Computer Security – ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, ch. Bleichenbacher's Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption, 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 *Proceedings of the 22nd ACM SIGSAC 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., Pokornỳ, 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] Martin, D. Secure protocols in a hostile world. Bristol Cryptography Blog, Sept. 2015. http://bristolcrypto.blogspot.co.il/2015/09/secure-protocols-in-hostile-world.html.

[33] MAVROGIANNOPOULOS, N., VERCAUTEREN, F., VELICHKOV, V., AND PRENEEL, B. A cross-protocol attack on the TLS protocol. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 62–72.

[34] MAYER, W., ZAUNER, A., SCHMIEDECKER, M., AND HUBER, M. No need for black chambers: Testing TLS in the e-mail ecosystem at large. *CoRR abs/1510.08646* (2015).

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

[36] 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.

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

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

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

[40] RISTIĆ, I. https://www.trustworthyinternet.org/ssl-pulse/.

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

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

[43] STEUBE, J. Optimizing computation of hash-algorithms as an attacker. In *Passwords* (Las Vegas, 2013). http://hashcat.net/events/p13/js-ocohaaaa.pdf.

[44] STEVENS, M., SOTIROV, A., APPELBAUM, J., LENSTRA, A., MOLNAR, D., OSVIK, D. A., AND WEGER, B. *Advances in Cryptology - CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ch. Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate, pp. 55–69.

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

[46] WAGNER, D., AND SCHNEIER, B. Analysis of the SSL 3.0 protocol. *The Second USENIX Workshop on Electronic Commerce Proceedings* (1996).

[47] WANG, X., AND YU, H. How to break MD5 and other hash functions. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques* (Berlin, Heidelberg, 2005), EUROCRYPT'05, Springer-Verlag, pp. 19–35.

[48] YOO, A. B., JETTE, M. A., AND GRONDONA, M. Slurm: Simple Linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing* (2003), Springer, pp. 44–60.

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

A Public key reuse

Reuse of RSA keys among different services was identified as a huge amplification to the number of services vulnerable to DROWN. Table 5 describes the number of reused RSA keys among different protocols. The two clusters 110-143 and 993-995 stick out as they share the majority of public keys. This is expected, as most of these ports are served by the same IMAP/POP3 daemon. The rest of the ports also share a substantial fraction of public keys, usually between 21% and 87%. The numbers for HTTPS (port 443) differ as there are four times as many public keys in HTTPS as in the second largest protocol.

B Adaptations to Bleichenbacher's attack

B.1 Calculating the success probability of a fraction

For a given fraction u/t, we can compute the probability of success with a randomly chosen TLS conformant ciphertext. Let $m_1 = m_0 \cdot u/t = m_1[1]||...||m_1[\ell]$ - i.e. $m_1[i]$ is the ith byte of m_1. Let k be the fixed byte length of the oracle response. 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 randomly generated m_0, this condition holds with probability $1/t$.
2. $m_1[1] = 0$ and $m_1[2] = 2$, or the integer $m \cdot u/t \in [2B, 3B-1)$. For a randomly generated m_0 divisible by t and for a given fraction u/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. $\forall i \in [3, \ell-(k+1)], m_1[i] \neq 0$, or all bytes between the first two bytes and the $(k+1)$ least significant bytes are non-zero. This condition holds with probability $(1-1/256)^{\ell-(k+3)}$.
4. $m_1[\ell-k] = 0$, or the $(k+1)$st least significant byte is 0. This condition holds with probability $1/256$.

As an example, let us assume a 2048-bit RSA ciphertext with $k = 5$, and consider the fraction $u = 7, t = 8$. We have

$$\begin{aligned} P(t|m_0) &= 1/t = 1/8 \\ P(m_1[1,2] = 00||02 \,|\, t|m_0) &= 0.71 \\ P(\forall i \in [3, \ell-6]\, m_1[i] \neq 0) &= (1-1/256)^{248} = 0.37 \\ P(m_1[\ell-5] = 0) &= 1/256 \end{aligned}$$

The overall probability of success is $P = 1/8 \cdot 0.71 \cdot 0.37 \cdot 1/256 = 1/7,774$; thus we expect to find an SSLv2 conformant ciphertext after testing 7,774 randomly chosen

Port	25 (SMTP)	110 (POP3)	143 (IMAP)	443 (HTTPS)	465 (SMTPS)	587 (SMTP)	993 (IMAPS)	995 (POP3S)
25	1,115 (100%)	331 (32%)	318 (32%)	196 (4%)	403 (47%)	307 (48%)	369 (33%)	321 (32%)
110	331 (30%)	1,044 (100%)	795 (79%)	152 (3%)	337 (39%)	222 (35%)	819 (72%)	877 (87%)
143	318 (29%)	795 (76%)	1,003 (100%)	149 (3%)	321 (38%)	220 (35%)	878 (78%)	755 (75%)
443	196 (18%)	152 (15%)	149 (15%)	4,579 (100%)	129 (15%)	94 (15%)	175 (16%)	151 (15%)
465	403 (36%)	337 (32%)	321 (32%)	129 (3%)	857 (100%)	463 (73%)	396 (35%)	364 (36%)
587	307 (28%)	222 (21%)	220 (22%)	94 (2%)	463 (54%)	637 (100%)	259 (23%)	229 (23%)
993	369 (33%)	819 (78%)	878 (88%)	175 (4%)	396 (46%)	259 (41%)	1,131 (100%)	859 (85%)
995	321 (29%)	877 (84%)	755 (75%)	151 (3%)	364 (42%)	229 (36%)	859 (76%)	1,010 (100%)

Table 5: **Impact of key reuse across ports.** Number of shared public keys among two ports, in thousands. Each column states what number and percentage of keys from the port in the header row are used on other ports. For example, 18% of keys used on port 25 are also used on port 443, but only 4% of keys used on port 443 are also used on port 25.

TLS conformant ciphertexts. We can decrease the number of TLS conformant ciphertexts needed by multiplying each candidate ciphertext by several fractions.

B.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, which relies on the fractional multipliers used in the first step. Suppose the attacker uses a fraction $u/t = 8/7$ to compute a new SSLv2 conformant candidate, and suppose that m_0 is indeed divisible by $t = 7$. This implies that 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 the attacker to reduce 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 saves us a factor of $v_2(u)$ in the required encryption attempts. 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 $1/P_F \cdot |F|$: For each ciphertext, we need to query the oracle with each fraction. Accordingly, the required number connections to the server is $2 \cdot 1/P_F \cdot |F|$, since as explained earlier each logical query consists of two connections to the server.

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 $E_f = E_{u/t} = 2^{40-v_2(u)}$. If we further define the required encryption attempts when testing a single ciphertext with each fraction from a given set of fraction F as $E_F = \sum_{f \in F} E_f$ then the required number of encryption attempts throughout the attack for a given set of fractions is $(1/P_F) \cdot E_F$.

Using this approach, 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 by using the monte-carlo estimation technique described above, with one billion random ciphertexts per tested fraction set F.

B.3 Efficiently computing rotations and multipliers

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 $\ell - k - 3$ bytes of m_2 are all nonzero is about 0.37 as in the previous section, and the probability that the $k+1$ least significant delimiter byte is `0x00` is 1/256. Thus a randomly chosen s will work with probability $2^{-25.4}$ and we expect to need to try $2^{25.4}$ values of s before succeeding.

However, since we have already learned $k+3$ most significant bytes of $m_1 \cdot R^{-1} \bmod N$, for $k \geq 4$ and $s < 2^{30}$ we do not need to query the oracle to learn if the two most significant bytes are SSLv2 conformant; we can compute this ourselves from our knowledge of $\tilde{m_1} \cdot R^{-1}$.

We could simply iterate through values of s, test that the top two bytes of $\tilde{m}_1 \cdot R^{-1} \bmod N$ are SSLv2 conformant, and only query the oracle $\mathcal{O}$ for values of s that satisfy this test; this means that for our 2048-bit modulus we expect to test 2^{16} values offline per oracle query. The probability that our query is conformant is then $P = (1/256) * (255/256)^{249} \approx 1/678$ so we 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 $XZ < N$. $N \approx 2^{8\ell}$, 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.

B.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 this phase 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 $2^{8\ell-66}$. Then 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 is at most $2^{8\ell-66}$, with high probability for any $s_3 < 2^{30}$, $m_3 \cdot s_3$ lies in an interval whose length is at most $2^{8\ell-36} < B$, so we know the two most significant bytes of $m_3 \cdot s_3$. Furthermore, we know the exact value of the 6 least significant bytes even after multiplication. So we test possible values of s_3, and for values such that $m_3 \cdot s_3$ starts with the required 00 02 bytes, and the 6th least significant byte is zero, we query the oracle as to the validity of $c_3 \cdot s_3^e \bmod N$. The only condition for PKCS conformance which we haven't verified before querying the oracle is

$$\forall i \in [3, \ell - 6], (m_3 \cdot s_3)[i] \neq 0$$

which holds with probability 0.37. So after roughly $1/0.37 = 2.72$ queries, we expect to get a positive answer from the oracle.

Since we know the value of the 6 least significant bytes after multiplication, there's no component of breaking a symmetric cipher here - if the message is SSLv2 conformant after multiplication, we know the symmetric key, and can test whether it fits the received `ServerVerify` message.

B.5 General DROWN Bleichenbacher iterations

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 00 02 bytes once in roughly every two queries on average, and requires the number of queries to be roughly double the number of bits in the RSA modulus. 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 cause the 6th least significant byte to be zero, and we don't need to break a symmetric key since we know the value of the 5 least significant bytes. However, we cannot ensure that the padding is non-zero when querying—we simply hope that is the case, which as usual happens with probability 0.37.

Therefore, for a 2048-bit modulus, the overall expected number of queries for this phase is roughly $2048 * 2/0.37 = 11,000$. This is indeed the average number of queries we require in practice when running our implementation of the attack.

B.6 General DROWN attack performance

For a given set of fractions, F, the required number of recorded client connections A is a random variable distributed geometrically with a success probability $P = P_F$. For typical fraction sets, $1/13,000 < P_F < 1/600$. The required number of Bleichenbacher queries against the target server during the first step of the attack is a random variable, B, such that $B = |F| \cdot A$. As each query consists of two separate connections to the target server, the required number of connections is always twice the number of queries. And last, the required keys to be tested overall is another random variable $C = k_F \cdot B; k_F \approx 2^{40}$.

Summing the figures from the different phases for a 2048-bit RSA modulus, the attack requires in expectation $13,838 + 1,393 + 1,393 + 6 + 22,140 = 38,770$ connections to the target server, when optimizing for the number of queries in phase 1. Each oracle query requires two connections to the server.

Re-calculating the numbers for a 1024 bit modulus, the primary element that needs to change is $P_1 = P(\forall i \in [3, \ell - 6] : m_i \neq 0) = (1 - 1/256)^{120} = 0.62$, which appears in phases 1, 2, 3 and 5. For phase 5, the number of queries is now in expectation $1024 * 2/0.62 = 3,303$. The total expected number of server connections is therefore $8,258 + 826 + 826 + 6 + 6,606 = 16,522$, again when optimizing for the number of queries in phase 1.

Similarly, re-calculating the numbers for a 4096 bit modulus, $P_1 = (1 - 1/256)^{504} = 0.14$, and the number of queries in phase 5 is now roughly $4096 * 2/0.14 = 58,514$. The algorithm for phase 5 can be further optimized if that is the case of interest; we omit these optimizations for space reasons. Again, summing up yields $36,571 + 3,657 + 3,657 + 29 + 117,028 = 160,942$ required connections to the server.

B.7 Special DROWN attack performance

In the first step, we can use the same fraction analysis as before. 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 we now only need 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 at this point could directly 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 for the attacker to apply a rotation 10 more times to recover the remaining bits of the plaintext. 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 $\mathcal{O}_{\mathsf{SSLv2\text{-}extra\text{-}clear}}$ oracle, the attacker can increase his chances of success by querying the server multiple times per ciphertext and fraction, using different cipher suites with different key lengths. He can query 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.

C Highly optimized GPU implementation

The most computationally expensive part of our general DROWN attack is breaking the 40-bit symmetric key. We wanted to find the platform that would have the best tradeoff of cost and speed for the attack, so we performed some preliminary experiments comparing performance of symmetric key breaking on CPUs, GPUs, and FPGAs. These experiments used a naïve version of the attack using the OpenSSL implementation of MD5 and RC2.

The CPU machine contained four Intel Xeon E7-4820 CPUs with a total of 32 cores (64 concurrent threads). The GPU system was equipped with a ZOTAC GeForce GTX TITAN and an Intel Xeon E5-1620 host CPU. The FPGA setup consisted of 64 Spartan-6 LX150 FPGAs.

We benchmarked the performance of the CPU and GPU implementations over a large corpus of randomly generated keys, and then extrapolated to the full attack. For the FPGAs, we tested the functionality in simulation and estimated the actual runtime by theoretically filling the FPGA up to 90% with the design, including communication. Table 6 compares the three platforms.

While the FPGA implementation was the fastest in our test setup, the speed-to-cost ratio of GPUs was the most promising. Therefore, we decided to focus on optimizing the attack on the GPU platform. We developed several optimizations:

Platform	Hardware	Cost	Full attack	Cost to perform attack in 1 day
Naïve CPU	4 Intel Xeon E7-4820	$21,400	114 days	$2,440,000
Naïve GPU	ZOTAC GeForce GTX TITAN	$2,400	189 days	$450,000
Naïve FPGA	64 Spartan-6 LX150	$60,000	51.5 days	$3,090,000
Optimized Hashcat	NVIDIA GTX / AMD R9	$18,040	0.75 days	$13,500
Optimized EC2	NVIDIA	$440	0.33 days	$147

Table 6: **Time and cost efficiency of our attack on different hardware platforms.** The brute force attacks against symmetric export keys are the most expensive part of our attack. We compared the performance of a naïve implementation of our attack on different platforms, and decided that a GPU implementation held the most promise. We then heavily optimized our GPU implementation, obtaining several orders of magnitude in speedup.

Generating key candidates on GPUs. Our naïve implementation generated key candidates on the CPUs. For each hash computation, a key candidate was transmitted to the GPU, and the GPU responded with the key validity. The bottleneck in this approach was the PCI-E Bus. Even newer boards with PCI-E 3.0 or even PCI-E 4.0 are too slow to handle the large amount of data required to keep the GPUs busy. We solved this problem by generating the key candidates directly on the GPUs.

Generating memory blocks of keys. Our hash computation kernel had to access different candidate keys from the GPU memory. Accessing global memory is typically a slow operation and we needed to keep memory access as minimal as possible. Ideally we would be able to access the candidate keys on a register level or from a constant memory block, which is almost as fast as a register. However, there are not enough registers or constant memory available to store all the key values.

We decided to divide each key value into two parts k_H and k_L, where $|k_H| = 1$ byte and $|k_L| = 4$ bytes. We stored all possible 2^8 k_H values in the constant read-only memory, and all possible 2^{32} k_L values in the global memory. Next we used an in-kernel loop. We loaded the latter 4 bytes from the slow global memory and stored it in registers. Inside the inner loop we iterated through our first byte k_H by accessing the fast constant memory. The resulting key candidate was computed as $k = k_H || k_L$.

Using 32-bit data types. Although modern GPUs support several data types ranging in size from 8 to 64 bits, many instructions are designed for 32-bit data types. This fits the design of MD5 perfectly, because it uses 32-bit data types. RC2, however, uses both 8-bit and 16-bit data types, which are not suitable for 32-bit instruction sets. This forced us to rewrite the original RC2 algorithm to use 32-bit instructions.

Avoiding loop branches. Our kernel has to concatenate several inputs to generate the `server_write_key` needed for the encryption as described in Section 2.2. Using loops to move this data generates branches because there is always an if() inside a for() loop. To avoid these branches, which always slow down a GPU implementation, we manually shifted the input bytes into the 32-bit registers for MD5. This was possible since the hash computation inputs, $(mk_{clear} || mk_{secret} ||\text{“0”}|| r_c || r_s)$, have constant length.

Optimizing MD5 computation. Our MD5 inputs have known input length and block structure, allowing us to use the so-called zero-based optimizations. Given the known input length (49 bytes) and the fact that MD5 uses zero padding, in our case the MD5 input block included four 0x00 bytes. These `0x00` bytes are read four times per MD5 computation which allowed us to drop in total 16 ADD operations per MD5 computation. In addition, we applied the Initial-step optimizations used in the Hashcat implementation [43].

Skipping the second encryption block. The input of the brute-force computation is a 16-byte client challenge r_c and the resulting ciphertext from the `ServerVerify` message which is computed with an RC2 cipher. As RC2 is an 8-byte block cipher the RC2 input is split into two blocks and two RC2 encryptions are performed. In our verification algorithm, we skipped the second decryption step as soon as we saw the key candidate does not decrypt the first plaintext block correctly. This resulted in a speedup of about a factor of 1.5.

RC2 permutation table in constant memory. The RC2 algorithm uses a 256-byte permutation table which is constant for all RC2 computations. Hence, this table is a good candidate to be put into the constant memory, which is nearly as fast as registers and makes it easy to address the table elements. When finally using the values, we copied them into the even faster shared memory. Although this copy operation has to be repeated, it still led to a speed up of approximately a factor of 2.

RC2 key setup without keysize checks. The key used for RC2 encryption is generated using MD5, thus the key size is always 128 bits. Therefore, we do not have to check for the input key size, and can simply skip the size verification branch completely.

D Amazon EC2 evaluation

Amazon EC2 billing is based on the *instance-hour*. An *instance* represents a single virtualized machine and its associated cores, memory, and storage. For our experiments we used `g2` instances, which are equipped with high-performance NVIDIA GPUs, each with 1,536 CUDA cores. The two available models for this instance type are the `g2.2xlarge` and the `g2.8xlarge`, containing one and four GPUs, respectively.

It is possible to request instances at a fixed on-demand rate, or bid on instances at the discounted spot instance rate. Spot instances may be terminated depending on demand, but the savings in cost are significant compared to the on-demand rate. When we ran our experiments in January 2016, the on-demand rate for the `g2.2xlarge` model was $0.65/hr and the rate for the `g2.8xlarge` model was $2.65/hr, while the average spot rates we paid were $0.09/hr and $0.83/hr respectively.

We used a cluster composed of 200 spot instances: 150 `g2.2xlarge` which contain one GPU and 50 `g2.8xlarge`, each containing four GPUs, spread across multiple availability zones within the US-East region. This distribution was determined by price: we were not able to launch more than 50 `g2.8xlarge` instances without a sharp spike in spot prices. We used the optimized Hashcat implementation on the same workload of key requests as the experiments run on the Hashcat servers. We used Slurm [48] to distribute jobs across compute nodes.

The GPU breaking experiment completed successfully, with two minor caveats. First, the 150 `g2.2xlarge` nodes completed their workloads at the 6h26m mark, while the other 50 `g2.8xlarge` nodes did not finish until the 7h41m mark. More careful job distribution would ensure that all nodes completed at approximately the same time, reducing the overall runtime. Second, in this particular run, 7.2% of the jobs that we expected to complete were terminated early due to overheating GPUs. The attack was successful despite the failed jobs, so we did not rerun them. In a more carefully engineered implementation, the unfinished jobs could have been reallocated to the unused GPU capacity without increasing the overall runtime.

The total cost of the experiment was $440, and terminated in under 8 hours including startup and shutdown.

E A brief history of obsolete cryptography

A flaw was first observed in the MD5 hash function in 1996; the first collision was discovered in 2004 [47], but MD5 was still in use by certificate authorities in 2009 when Stevens et al. [44] used a chosen-prefix MD5 attack to construct a malicious TLS certificate with a valid CA signature. The RC4 stream cipher was observed to be biased as early as 1995 and shown to be catastrophically broken in the context of WEP in 2001 [18]; it was used by about 50% of TLS connections in 2013 when AlFardan et al. demonstrated near-practical attacks against RC4 in TLS [3]. TLSv1.0 was standardized in 1998 to replace SSLv3; before the POODLE attack [37] was shown to render all SSLv3 block cipher suites insecure in 2014, support for SSLv3 was near 100% for popular HTTPS sites, and most clients were vulnerable to a downgrade attack from TLS to SSLv3 [40]. Export-grade cipher suites for TLS have been obsolete since 2000, when the United States relaxed restrictions on commercial and open source software; before the FREAK attack [6] demonstrated widespread implementation flaws allowing a catastrophic downgrade attack exploiting export RSA, 37% of HTTPS sites with browser-trusted certificates supported export-grade RSA. Three months later the Logjam attack [1] demonstrated a TLS protocol flaw downgrade attack exploiting export Diffie-Hellman; 8.4% of the Alexa top million sites were vulnerable at the time.