Open to a fault: On the passive compromise of TLS keys via transient errors

George Arnold Sullivan, *University of California, San Diego;* Jackson Sippe, *University of Colorado Boulder;* Nadia Heninger, *University of California, San Diego;* Eric Wustrow, *University of Colorado Boulder*

https://www.usenix.org/conference/usenixsecurity22/presentation/sullivan

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open to a fault: On the passive compromise of TLS keys via transient errors

George Arnold Sullivan
UC San Diego

Jackson Sippe
CU Boulder

Nadia Heninger
UC San Diego

Eric Wustrow
CU Boulder

Abstract

It is well known in the cryptographic literature that the most common digital signature schemes used in practice can fail catastrophically in the presence of faults during computation. We use passive and active network measurements to analyze organically-occuring faults in billions of digital signatures generated by tens of millions of hosts. We find that a persistent rate of apparent hardware faults in unprotected implementations has resulted in compromised certificate RSA private keys for years. The faulty signatures we observed allowed us to compute private RSA keys associated with a top-10 Alexa site, several browser-trusted wildcard certificates for organizations that used a popular VPN product, and a small sporadic population of other web sites and network devices. These measurements illustrate the fragility of RSA PKCS#1v1.5 signature padding and provide insight on the risks faced by unprotected implementations on hardware at Internet scale.

1 Introduction

During 2009-2011, Toyota issued multiple vehicle recalls after hundreds of crashes had been reported relating to unintended acceleration. Initially, Toyota placed the blame on driver error, shifting floor mats, and sticky accelerator pedals. In 2013 expert witness Michael Barr testified in the Bookout v. Toyota Motor Corp. case that a single bit flip sufficed to kill a throttle monitoring task, resulting in uncontrolled acceleration [4, 55]. Toyota lost the case and began settling with crash victims out of court.

The exact cause of the memory corruption in Toyota vehicles was never established: it could have been buffer overflows, cosmic rays, or hardware faults. No matter the underlying cause, the existing hardware protections were insufficient, and the software was brittle in the face of hardware errors.

Cryptographic software engineering is—fortunately—less often considered to be a matter of life or death. Nonetheless, faults can have similarly catastrophic impact on cryptographic systems. As prior work has shown, attacker-induced or naturally occurring bit flips can corrupt cryptographic computations, causing them to produce incorrect results, or even leak secret information or keys [12].

In this paper, we show that these attacks can be applied *entirely passively*, allowing a network adversary to derive TLS RSA private keys by observing network traffic. When errors occur[1] during a server's RSA signature computation, the resulting failed handshake can give an attacker sufficient information to derive the server's long-term private key.

We demonstrate these attacks by collecting 5.8 billion TLS handshakes from two different university networks. These handshakes included 3.3 billion connections using TLS 1.2 or below and 2.7 billion server signatures. Over a few months, we found nearly 2,000 non-validating digital signatures from failed handshakes. Some of these failed handshakes allowed us to compute three RSA private keys associated with Baidu, a multinational technology company and top 10 Alexa site. These keys were used to secure more than a million connections to hundreds of hosts in our dataset corresponding to dozens of Baidu's cloud-based services.

This passive attack is particularly concerning in the context of nation-state adversaries conducting mass surveillance. Unlike active attacks or remote compromise, which risk leaving evidence of tampering, passive fault analysis leaves no trace on either client or server. A network adversary only need observe network traffic passively and perform simple cryptographic calculations, capabilities that modern nation states are known to posses and employ for the purposes of network surveillance. This attack is exacerbated when TLS servers and clients negotiate non-forward secure ciphers, allowing the network attacker to passively decrypt encrypted TLS payloads using the server's private key, without leaving any trace of compromise.

In addition to demonstrating passive fault attacks, we also carried out active scans of TLS hosts, and performed a retrospective analysis of historical TLS scan data between 2015 and 2022 that included tens of thousands of non-validating

[1] perhaps due to bit flips, faulty hardware, memory corruption, or disk failure

signatures. In total, we computed 127 private keys from these active scans.

We compare our results to active scans from a 2015 technical report by Weimer [53], who appears to have been the first to observe that active scans could be used to detect or trigger these types of RSA signature faults at scale. He found that several open-source TLS libraries did not implement countermeasures against signature faults, and performed active TLS scans over a period of months that resulted in a few hundred invalid signatures that successfully compromised private keys, mostly from devices from several vendors.

Our passive analysis and recent active scans show that these problems are still present in current implementations. We were able to compute the browser-trusted private keys for a handful of user-facing web sites from sporadic faults, as well as observing dozens of certificate private keys compromised by devices. These certificates span from untrusted device default certificates to CA-signed browser-trusted wildcard certificates for entire organizations. Although all of the open source libraries we inspected have implemented countermeasures, it appears some proprietary TLS implementations are still vulnerable to this attack.

From signature faults to key recovery. The flaw we exploit is well known in the cryptographic side channel literature. Boneh, DeMillo, and Lipton describe in their 1997 paper [11] an RSA key recovery attack: Almost all RSA implementations use the Chinese Remainder Theorem (CRT) optimization for modular exponentiation in RSA signing, but errors that occur in one of the half-exponentiations in this algorithm can result in leaking information that can be used to derive the RSA private key. Lenstra improved the attack to require only one signature when the message is known [12, 39]. This attack works against any deterministic RSA signature scheme using the CRT optimization. The countermeasure to these attacks is to validate the RSA signature before sending it.

Prior to Weimer's work in 2015 [53], almost no implementations validated RSA signatures before sending. Following the report, all of the software TLS libraries Weimer contacted implemented countermeasures, and Cavium issued a patch for their cryptographic accelerators, which appeared to be at fault for several of the vulnerable devices he discovered.

In this work, we find that spontaneous faults compromising RSA keys through PKCS#1v1.5 signatures continue to be present as a low but persistent rate in both passive and active network measurements over time, despite the attention drawn to this vulnerability in 2015 [53]. In the present era in 2022, we find that this flaw is not just present in the types of network devices that have already been observed to suffer from cryptographic implementation flaws in previous studies [29, 30, 53], but that it also affects user-facing websites and infrastructure that receive significant amounts of network traffic.

These vulnerabilities are due to a hazardous combination of cryptographic libraries vulnerable in the face of computational errors, and the brittle nature of the RSA PKCS#1v1.5 signature padding scheme as used in TLS 1.0–1.2. PKCS#1v1.5 signature padding makes key compromise trivial in the presence of CRT faults. Prior to TLS 1.3, handshakes take place in plaintext, providing all the information a passive network adversary needs to validate signatures on observed connections, or derive keys when errors occur.

On a more positive note, TLS 1.3 provides multiple countermeasures against these issues, including moving the key exchange earlier in the handshake in order to ensure that certificates and signatures are sent encrypted, and using RSA-PSS signature padding, which prevents this type of key compromise if implemented correctly.

Contributions Our contributions are as follows:

- We study the first passively-collected dataset of 2.7 billion server-generated TLS signatures on real-world network traffic from two universities.
- We investigate the impact of hardware faults in several signature schemes in practice, including RSA PKCS#1 v1.5, RSA-PSS, and ECDSA
- We demonstrate both the feasibility and the impact of passively monitoring for cryptographic hardware faults.
- We perform a retrospective analysis of historical TLS scan data and supplement with current TLS scan data to examine the impact of this vulnerability over time.
- We discuss defenses to this attack at the application, library, and protocol level.

Ethical considerations

Passive measurement of real network data invokes natural privacy concerns. The network measurement and processing infrastructure we use was developed in consulation with our campus institutional review boards (IRBs) and network operations and cybersecurity staff. The tap infrastructure and data processing protocols we use have been evaluated as exempt by our respective IRBs.

We have taken several steps to minimize the potential exposure of personally identifiable information from the human network users whose data we analyze. Campus IP addresses and MAC addresses that are present in the packets captured by our infrastructure are discarded during parsing and before storage. At UCSD, these fields are encrypted with a keyed format-preserving encryption scheme, and at CU Boulder we chose not to log source IP addresses at all. All content data was discarded: we parse only the cryptographic handshakes. In any case, because we focus on TLS, the content is all encrypted. We did not attempt to deanonymize any connections or decrypt any traffic using the private keys we computed.

The active scans that we performed were rate-limited to avoid excessive burden on the hosts we connected to, and we followed best practices for active network scanning including scanning from an identifiable host with information on our project, and maintaining an opt-out blocklist.

Disclosure

We disclosed to Baidu on January 26, 2022 and worked with their team to verify the exposure. Baidu has acknowledged the problem and corrected the issue. They updated their keys in late February, 2022. We disclosed to Xerox on May 6, 2022 and have received an acknowledgement. We disclosed a historical vulnerability to doi.gov on May 11, and have received a confirmation that the vulnerable site has a new certificate and is no longer public facing. We disclosed to Cisco on May 13, 2022, and have received an acknowledgement. We communicated with Microsoft on May 21, 2022 and with Alibaba Cloud on May 22, 2022. We have included more information as available below.

2 Background and related work

RSA public keys remain overwhelmingly popular for signature usage in TLS; in our passive dataset around 90% of TLS connections used an RSA signature. (See Table 3.)

2.1 RSA signatures

A textbook RSA public key is a pair of integers (N,e) where N is a public modulus that is the product of two equal-sized primes p and q, and e is the public signature verification exponent, nearly always set to 65537 in TLS. The private key is the pair (N,d) where $d = e^{-1} \bmod (p-1)(q-1)$ is the secret signing exponent.

A textbook RSA signature s on a message m is the value $s = m^d \bmod N$. To verify a textbook RSA signature, the verifier checks whether $m \overset{?}{=} s^e \bmod N$.

2.1.1 Padding and Optimizations

Textbook RSA signatures as described above are vulnerable to a number of elementary signature forgery attacks [13], so RSA signatures are generally padded and hashed.

PKCS#1v1.5 The most common RSA signature padding scheme historically has been PKCS#1v1.5 padding [37]. For signatures, this padding takes the form

`pad(m) = 00 || 01 || FF .. FF || ASN.1 || H(m)`

where `ASN.1` is an algorithm identifier specifying the hash function used in the signature, and `H(m)` is a cryptographic hash of the message m using the specified hash function. In typical modern usage, N is 2048 bits, and H is SHA-256 or SHA-512. The length of the `ASN.1` algorithm identifier, null spacing bytes, and ASN.1 length encoding for the hash is 20 bytes. The `FF` bytes occupy the remaining space to fill the padded message to the byte length of the modulus N. The value `pad(m)` is exponentiated and validated as in textbook RSA described above.

RSA PKCS#1v1.5 signature padding has been proven to be secure in the random oracle model [32]. (This is in contrast to the case of RSA PKCS#1v1.5 encryption padding, which is not secure against chosen ciphertext attacks [8] and has resulted in numerous real-world vulnerabilities [3, 10].)

RSA-PSS RSA-PSS is an alternative randomized RSA signature padding scheme [5]. A modified version of RSA-PSS (often referred to as RSASSA-PSS) appears in PKCS#1v2.1 [35] and 2.2. Unlike PKCS#1v1.5 padding, PSS padding is randomized using a salt. The input to RSASSA-PSS is a message hash $mHash$, which is then hashed together with padding and the salt s to obtain a string H. H is then processed through a *mask generating function* MGF and xored with padding and the salt value to obtain a value $maskedDB$. The encoded message to which RSA signing is applied is then

`pad(mHash) = maskedDB || H || BC`

where `BC` is the hexadecimal byte value. To verify the signature, the verifier must know the message hash that is being signed. The verifier inverts RSA signing, parses `pad(mHash)` from the padded message, recovers the salt, and then verifies the salted hash H.

TLS 1.3 specifies the use of PSS padding for RSA. However, historically RSA-PSS has remained less popular than PKCS#1v1.5 padding. Gutmann has criticized standardizations of PSS as potentially being vulnerable to parameter substitution and malleability attacks [28].

Although PSS is typically described as randomized, PKCS#1v2.1 contains a note that “randomness is not critical to security” for PSS, and suggests that “In situations where random generation is not possible, a fixed value or a sequence number could be employed instead, with the resulting provable security similar to that of FDH” (Full-Domain Hashing [5]).

CRT Implementations often use the Chinese Remainder Theorem (CRT) to speed up the modular exponentiation computations required for signing. To do this, an implementation computes the CRT secret exponents $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, and the coefficient $q_{inv} = q^{-1} \bmod p$. Then the value $s = m^d \bmod N$ can be computed by first computing the half-exponentiations $s_p = m^{d_p} \bmod p$ and $s_q = m^{d_q} \bmod q$ then recovering the full s using Garner's formula: $s = s_p q q_{inv} + s_q(1 - q q_{inv}) = (s_p - s_q) q q_{inv} + s_q \bmod N$.

Modular exponentiation The simplest efficient modular exponentiation algorithm is the binary square-and-multiply algorithm. However, textbook square-and-multiply is one of the canonical examples of an algorithm that is vulnerable to side-channel attacks, and modern implementations typically adopt multiple optimizations over textbook square-and-multiply [26], as well as using countermeasures against side-channel attacks. For instance, to protect against side-channel attacks, implementations may blind the exponent (by adding multiples of $\phi(N)$ for the full exponent, or $p-1$ or $q-1$ for the half-exponents) or message using RSA blinding.

In Section 4.2 we discuss some of the implementation choices made by popular TLS libraries in more detail. Message blinding appears to be near-ubiquitous; exponent blinding less so.

2.2 Fault Attacks against RSA

CRT-RSA RSA is vulnerable to an elementary key recovery attack if a fault occurs during the signing process in an implementation that uses RSA-CRT, originally due to Lenstra [12, 39]. Let m be the padded and hashed message. Then a CRT-based signing implementation would compute two half-exponentiations $s_p = m^{d_p} \bmod p$ and $s_q = m^{d_q} \bmod q$ and use the CRT to recover the signature s.

Consider the scenario where a fault occurs during the computation of one of the half-exponentiations, and the resulting faulty signature s_f does not validate. However, s_f has the property that $s_f \equiv s_p \bmod p$ while $s_f \neq s_q \bmod q$. Then $\gcd(s_f^e - m, N) = p$, revealing the factorization of N.

This attack requires that the attacker have access to one faulty signature, and for the attacker to know the padded message that was signed. For RSA PKCS#1v1.5, the deterministically padded and hashed message is easy for an attacker to construct from known hash inputs. In the case of TLS 1.2 and below, the hash inputs are all parts of the handshake visible to either a passive observer or active handshake participant.

Any deterministic or partially randomized RSA signature scheme used with CRT computations is potentially vulnerable [17]. It has been shown that randomized PSS padding is provably secure against random fault attacks [18].

Other RSA Fault Attacks Lenstra [39] and Boneh, DeMillo, and Lipton [12] also describe hypothetical key recovery attacks that could apply if transient faults occur during a non-CRT signing operation that uses a square and multiply or similar exponentiation algorithm to do exponentiation with the secret exponent d. These attacks both require a large number of faulty signatures and a specific error pattern that seems unlikely to arise given current modular exponentiation implementations and the ubiquity of blinding.

Beyond modular exponentiation, Brier et al. [16] developed a key recovery RSA fault attack exploiting injecting faults into the modulus during the CRT reconstruction step, rather than the modular exponentiation.

2.2.1 Practical fault attack demonstrations

Since the publication of the attacks described above, hardware-based fault attacks have been extensively studied, mainly focused on active attackers in the context of smartcard and other hardware implementations [2, 9, 38, 43].

In the software realm, the Rowhammer attack gave a plausible route for an attacker to induce bit flips in cryptographic computations. Razavi et al. [49] demonstrated a Rowhammer attack against RSA that involved flipping bits in the RSA public key modulus until the modulus was sufficiently composite to be easy to factor, and Weissman et al. [54] demonstrated a CRT-based Rowhammer fault attack against RSA.

Most similar to our work, Weimer [53] carried out active network scans looking for RSA-CRT faults in TLS signatures in 2015, and was able to factor more than 200 RSA keys, each of which was associated with a hardware network device. Of these keys, three were part of certificates that had been signed by certificate authorities. (Incidentally, we note that there is a significant overlap between the device manufacturers listed by Weimer as producing faulty RSA signatures and the device manufacturers listed by [29] as producing RSA moduli with repeated prime factors.)

2.3 ECDSA signatures

ECDSA (elliptic curve digital signature algorithm) [34] is a randomized digital signature scheme based on elliptic curves. (ECDSA is based on the structure of DSA, a digital signature algorithm based on the prime field discrete logarithm problem, but DSA is almost never used for TLS, so we omit a discussion here.)

An ECDSA public key consists of a set of curve parameters together with a public curve point. The curve parameters are almost always specified by naming a fixed, named curve over some fixed finite field along with its associated generating point G of a suitable group of points of prime order n. The most commonly seen curves are three curves standardized by NIST, called `secp256r1`, `secp384r1`, and `secp521r1`; the first is by far the most popular in TLS. The private key is an integer $0 < d < n$, and the public curve point is dG.

To generate an ECDSA signature, let h be the integer representation of the message hash h. The signer generates an ephemeral private key $0 < k < n$ which is often referred to as the *nonce*, and then generates the values $r = x(kG)$ and $s = k^{-1}(h + dr) \bmod n$ where the notation $x()$ means to take the x-coordinate of the curve point, and where the value of r is interpreted as an integer modulo n in the second equation. The signature is the pair (r, s).

ECDSA nonce generation ECDSA is well understood to be fragile if ever used with an insecure random number generator [14, 15, 30]. If an adversary ever learns the nonce k used to generate a single signature (r, s), it is simple to derive the long-term secret signing key d by solving the equation relating s and k for the secret d, assuming that the message hash h for the signature is known.

Even more problematic, if a signer ever reuses a nonce k to sign two different messages with the same private key, it is simple to recover k and thus the private signing key d. Let (r_1, s_1) and (r_2, s_2) be vulnerable signatures on message hashes h_1 and h_2 respectively, with $r_1 = r_2$. Then $k = (h_1 - h_2)/(s_1 - s_2) \bmod n$ and we can solve for d as above.

Because of this, many modern ECDSA implementations generate signature nonces deterministically from the secret signing key and the message to be signed [46].

These deterministic ECDSA nonce constructions will ensure that the private key will never be revealed by an insecure random number generator, but they turn out to be more fragile against fault attacks.

2.4 Fault attacks on ECDSA

Poddebniak et al. [45] describe efficient fault attacks against ECDSA that require at least two signatures with the same signature nonce, one correct and one with a fault.

Let (r, s) be an ECDSA signature as above on message hash h. If the same message hash is signed a second time and a fault happens during the computation of r but no errors occur with the deterministically computed signature nonce k, the algorithm may return the faulty signature (r_f, s_f) with $r_f \neq r$ and $s_f = k^{-1}(h + dr_f) \bmod n$. An attacker who observes both the valid signature (r, s) and faulty signature (r_f, s_f) can subtract s_f from s and rearrange to solve for $k = d(r - r_f)(s - s_f)^{-1} \bmod n$. This can then be substituted into the the relation defining s to obtain $d = h(s - s_f)(s_f r - s r_f)^{-1} \bmod n$.

2.5 TLS

The internet is currently going through an algorithm transition from TLS 1.2 to TLS 1.3. TLS 1.3 is a significant change to the TLS handshake compared to previous versions of SSL/TLS.

TLS can use RSA signatures in two main ways: first, *certificates* can contain RSA public keys and signatures to verify the identity of servers using a certificate authority (CA) public key. Second, each TLS connection can contain a *server key exchange* that carries a signature from the server's corresponding RSA key. We focus on the latter in this paper.

While certificate keys and associated certificate signatures are long-lived, the signatures found in the server key exchange message are generated *per-connection*. In TLS 1.2 and below, this signature is over the server parameters (that is, the (elliptic curve) Diffie-Hellman key exchange values) as well as the client and server random nonces that are sent as part of the client and server hello messages. Servers use their private key to generate a new signature as part of every handshake that negotiates an ephemeral Diffie-Hellman (DHE) or elliptic curve Diffie-Hellman (ECDHE) cipher suite. In TLS 1.2, clients can provide a list of supported signature algorithms (including padding formats) and hash functions using the signature_algorithm extension. The server chooses the signature algorithm and hash function from the client's list.

If a server computing the key exchange signature has an error during the signing process, or the packet containing a signature suffers a transmission error that is not caught by the TCP checksum, the signature as received by the client will fail to validate with the server's public key contained in the certificate. In this case, the client should send a *TLS Alert* message, with a code dependent on implementation.

For our purposes in this paper, TLS 1.2 and below are notable compared to other common network protocols because the server's signature is over data that is entirely visible to a passive network attacker: an observer needs only to collect the client hello, server hello, certificate, and server key exchange records in order to validate the signature. If the signature fails to verify, the passive observer would also expect to see the client send a TLS Alert and close the connection.

TLS 1.3. TLS 1.3 has made multiple major changes affecting the security of RSA signatures. In particular, TLS 1.3 adds a default extension to encrypt the handshake messages following the client and server hello messages, and mandates RSA-PSS signature padding for RSA signatures in the handshake, although PKCS#1v1.5 padding is still allowed for signatures in certificates. The server generates a signature to validate every handshake, but it is encrypted, so a passive adversary cannot observe the signature sent by the server. Our passive analysis is thus not able to use TLS 1.3 handshake data. However, it is still possible to collect and validate RSA-PSS signatures over TLS 1.3 handshakes using active scans.

SSH and IPsec also use server signatures for authentication, but a passive network observer will not be able to independently validate the signature because the signature contains data that a passive observer cannot easily compute, like the shared Diffie-Hellman secret.

3 Passive Data Collection

Passive network analysis has several advantages over active scans for measuring the prevalence of cryptographic flaws.

- **Scalability.** While network researchers commonly scan using 1 Gbps connections [21], network taps can easily be 40 or even 100 Gbps. In addition, active scanning for TLS signature flaws requires checking every signature, while a passive tap can rely on clients to do the check for them.
- **Realistic datasets.** Network tap traffic can observe servers that traditional active scanning would miss, such as firewalled hosts, non-public SNI values, sites not in popular lists, and IPv6 hosts that cannot be enumerated.

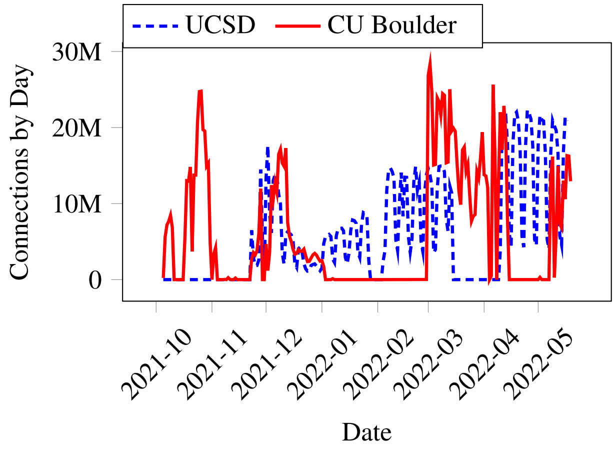

Figure 1: **TLS connections collected over time.** Our data collection process includes several idiosyncracies including downtime and gaps affecting the data we observe; holiday breaks and pandemic-related shutdowns have also affected connection patterns.

- **A shift in perspective.** Statistics from active network scans tend to be dominated by a large number of low-quality devices that are not representative of the hosts responsible for almost all real network traffic.
- **Modeling nation-state adversaries.** Nation states conducting mass surveillance are certainly capable of similar passive attacks, allowing them to decrypt traffic to or impersonate popular sites without leaving any evidence; our results demonstrate that a completely passive key compromise attack remains both feasible and practical even to less well resourced researchers, and was likely even more exploitable in the past.
- **Ethics.** Passive analysis avoids sending unnecessary traffic to servers, and instead relies on connections that are already made without our involvement.

3.1 Tap architecture

We collected data between October 2021 and May 2022. We used two network taps for data, one located at UCSD and one located at CU Boulder.

UCSD's tap receives a copy of campus wireless network mirrored from a campus Arista switch using two 10Gbps fiber lines. Due to limits in our processing capacity we exclude data from a number of large CDNs to reduce the load on our processing machine. These IPs account for about 1/3 of the traffic seen at UCSD. Once that data has been excluded we see a load of 4-6 Gbps on our server. To process this traffic we use PF_RING [40] and the Bro [42] Intrusion Detection System [2]. Our analysis includes data on any port that Bro identified as TLS traffic. To extract the TLS data we use custom Bro scripts to parse the TLS handshake messages and write the data out to logs, which are then written to a Hive database [23].

TLS Version	Connections	
SSL 3.0	207	0.00%
TLS 1.0	5.8M	0.16%
TLS 1.1	46K	0.00%
TLS 1.2	1.7B	47.50%
TLS 1.3	1.8B	49.04%
TLS 1.3-draft-26-fb	36.7M	1.00%
TLS 1.3-draft-23	449	0.00%
TLS 1.3-draft-28	33	0.00%
TLS 1.3-draft-26	24	0.00%
No version number	84.3M	2.29%
Total	3.7B	

Table 1: **Version numbers for SSL/TLS seen in UCSD connections.** The vast majority of the handshakes we observed were TLS 1.2 or TLS 1.3. "No version number" indicates that we did not see a server message containing a version number for that connection.

CU Boulder maintains a custom TLS packet analysis tool based on PF_RING [40] and adapted from TLS Fingerprint [25], running in a 1U server. This parser received a copy of all post-firewall traffic traversing the university network via a 40 Gbps fiber line. We analyze traffic on port 443.

TLS 1.2 Records We collected the following elements from each TLS 1.2 (or below) handshake:

The 32-byte client and server randoms, server name indication (SNI), server IP, client fingerprint [25] (a hash of identifying features from the Client Hello such as cipher suites and extensions in order to identify implementations while keeping specific devices anonymous), server certificate chain (which includes the server public key), and finally for connections using (EC)DHE cipher suites, the server key exchange parameters (Diffie-Hellman public values, signature algorithm, hash algorithm, and the server's signature).

Alert If an error occurs during the handshake, the client sends an Alert record to notify the server, and then closes the connection. Invalid signatures and other errors should produce a visible Alert record. We store the plaintext alert description.

TLS 1.3 Records TLS 1.3 connections perform an (elliptic curve or prime field) Diffie-Hellman key exchange, with keys sent in the Key Share extension in the Client and Server Hello messages. All records following the Server Hello are encrypted under the derived key, and not passively observable. This includes the certificate and server's signature authenticating the connection. Thus, for connections that negotiate TLS 1.3, we only collect the 32-byte client and server randoms, list of supported cipher suites (and the server selected one), supported and chosen protocol versions, and the client

[2] Now known as The Zeek Network Security Monitor: https://zeek.org/. However, our infrastructure is running an older version of Bro.

Description	UCSD		CU Boulder	
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	561M	41.6%	880M	63.2%
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	647M	48.0%	297M	21.4%
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	68M	5.0%	88M	6.3%
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	53M	4.0%	84M	6.1%
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	9.6M	0.7%	6.6M	0.5%
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	3.6M	0.3%	4M	0.3%
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	3.6M	0.3%	3.9M	0.3%
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	1M	0.1%	2.6M	0.2%
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	1.5M	0.1%	21	0.0%
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	160K	0.0%	300K	0.0%
Unknown	0	0.0%	6.4M	0.2%
Total	1.3B		1.4B	

Table 2: **Most common cipher suites for TLS 1.2 and below.** ECDHE key exchange with RSA signatures were the most popular public-key choices among cipher options.

and server Diffie-Hellman key exchange values.

3.2 Dataset

In total, we collected 3.3 billion TLS 1.2 and below connections across our datasets. Figure 1 shows the number of connections we collected over time for both UCSD and CU Boulder. Among these connections we observed 1.6 million unique server IPs at UCSD and 1.0 million at CU Boulder.

In our data, we see that the transition to TLS 1.3 is well under way, but not yet universal. As shown in Table 1, about half of connections are still being made with TLS 1.2 or below. More than two thirds of these connections in our datasets used RSA with PKCS#1v1.5 for the server key exchange signature. Table 2 lists the cipher suites we observed, with RSA signature cipher suites being overwhelmingly popular. Table 3 shows the breakdown of signature algorithms. The `rsa_pkcs1_*` signature algorithms represent between 65% and 70% of signatures we observed at each location.

Limitations Our collection tools both have limitations and bugs that occasionally cause us to miss some connections. This means there are idiosyncracies in our reported numbers, beyond the pre-existing biases present within any given institution. We report these limitations for completeness.

In particular, for CU Boulder,

- We do not parse connections that do not have a server key exchange message (That is, connections using RSA key exchange.)
- Before February 2022, we only parsed alerts from the client after the server key exchange.
- We do not parse TLS 1.0 handshakes.
- We began collecting TLS 1.3 handshakes in February 2022.
- We miss connections that have dropped or re-ordered packets in the handshake.
- Until February 2022, we dropped a large number of MTU-sized packets due to upstream limitations in our tap infrastructure.

Overall, these limitations mean that we are missing some data from about 80% of the TLS handshakes that pass through the CU Boulder network from October 2021 to January 2022, mainly due to a combination of TLS 1.3 handshakes and missing packets. After adding support for TLS 1.3, we miss data from around 50% of TLS handshakes. We report numbers from completed handshakes.

For UCSD, the limitations include:

- We filter out several large CDNs to reduce the traffic to a manageable size. This reduces our visibility into the very largest traffic sources on the network. [3]
- Approximately 3.3% of the 3.7 billion TLS connections we saw experienced some packet loss during collection.

3.3 Checking for anomalies

When a handshake encounters an error, the client or server would be expected to send a TLS alert containing a 1-byte description field. Table 4 shows the distribution of TLS Alerts seen throughout our dataset.

The presence of an alert in a connection signals that the client had some type of error. We are most interested in the case of a client who is unable to verify the signature on a server key exchange message. Overall, alerts happen in well under 1% of connections, with the most common alerts in our dataset being unrecognized CA, expired certificate, or unparsable messages. The specific alert that a client sends when encountering an invalid signature in the server key exchange varies across implementations. OpenSSL appears to send

[3] Our network analysis infrastructure at UCSD (which is shared with other researchers) couldn't keep up with 2x10 Gbps traffic without dropping packets. A fast IP filter dropping traffic from a few high-volume CDNs (including Akamai, Vodafone, RPS, megafon, and others) was the simplest solution that maximized parseable handshake data. These IPs were not excluded from collection at CU Boulder so we potentially would have seen evidence of vulnerabilities there.

Description	UCSD		CU Boulder	
rsa_pkcs1_sha256	521M	38.6%	421M	30.3%
rsa_pkcs1_sha512	366M	27.1%	548M	39.4%
rsa_pss_rsae_sha256	330M	24.5%	230M	16.6%
ecdsa_secp256r1_sha256	108M	8.0%	156M	11.2%
ecdsa_secp384r1_sha384	9M	0.7%	14M	1.0%
ecdsa_secp521r1_sha512	4.8M	0.4%	4.5M	0.3%
rsa_pkcs1_sha1	6.5M	0.5%	400K	0.0%
rsa_pkcs1_sha384	173K	0.0%	1M	0.1%
rsa_pkcs1_sha224	7.4K	0.0%	10K	0.0%
rsa_pss_rsae_sha512	5.7K	0.0%	875	0.0%
ecdsa_sha1	1.9K	0.0%	4.3K	0.0%
rsa_pss_rsae_sha384	490	0.0%	305	0.0%
Unknown	4M	0.3%	15M	1.1%
Total	1.3B		1.4B	

Table 3: **TLS 1.2 Signature Algorithms.** RSA PKCS#1v1.5 signatures were the most common in our dataset, followed by RSA-PSS and ECDSA.

Description	UCSD	CU Boulder
certificate_unknown	7.7M	1.1M
protocol_version	1.6K	2.6M
unknown_ca	1.1M	1.4M
handshake_failure	12K	1.1M
internal_error	797K	0
bad_certificate	15K	429K
certificate_expired	99K	99K
illegal_parameter	92K	65K
unrecognized_name	102K	0
unexpected_message	17K	11K
bad_record_mac	1.5K	4.6K
decrypt_error	585	5.1K
user_canceled	3.4K	0
record_overflow	0	3K
bad_certificate_status_response	2.1K	0
access_denied	418	1.1K
unsupported_certificate	13	1.4K
decode_error	170	125
Other	5	46
Unknown	11	0
Total Alerts	10M	6.9M
Total Connections	1.3B	1.4B

Table 4: TLS Alerts

the `decrypt_error` alert, which occurs in approximately 0.0002% of connections.

As noted in Limitations in Section 3.2, CU Boulder does not collect TLS alert messages sent by the server or those sent after the handshake. Therefore, UCSD sees a wider variety of alerts in Table 4 compared to CU Boulder.

4 Passive RSA Signature Analysis

In the UCSD data, we saw a total of 1.3 billion TLS connections where a server signature was sent. For every connection, we attempted to verify the signatures using the client random, server random, and server Diffie-Hellman parameters that we recorded for the connection. Among all these connections, we found 680 whose signatures did not verify. Of the connections that did not verify, 420 were signed using RSA, which was significantly more common than ECDSA in our dataset. See Table 3. Our CU Boulder data saw an additional 1.4 billion connections, with 1,306 invalid RSA signatures.

4.1 RSA PKCS#1v1.5 Signatures

Of the invalid signatures, 1,648 used RSA PKCS#1v1.5 signatures. For each faulty signature s_f, we examined the structure of the recovered "message" $m_f = s_f^e \bmod N$ after a modular exponentiation with the public key to try to classify the source of errors more carefully.

Figure 2 depicts a flowchart for how the structure of the faulty padded "message" m_f recovered during the RSA signature validation process may give some evidence for whether an invalid signature is due to errors in network transmission or during computation. Errors that occur in transmission between client and server would be expected to lead to an alert. We would not expect to see an alert if errors occurred in our data collection. In either case, transmission or collection errors cannot leak any private key information, while errors that occur during the server's computation of the signature may reveal the private key.

Correct PKCS#1v1.5 padding For 1,442 of the non-validating signatures, the recovered m_f had the correct padding format for a PKCS#1v1.5-padded signature—that is, they had the correct fixed padding bytes preceding the hash. That meant that these signatures failed to validate because the hash value did not match the recorded client and server randoms and server key exchange parameters.

This indicates that the problem with these signatures was not with the RSA calculation but rather with the hash function calculation either at signature generation or verification. We hypothesized that potential sources of error could be an error during the hash function calculation itself on the server, bit errors that evaded the TCP checksum during transmission, or corruption that happened after we collected the data.

Of the 1,442 faulty signatures in this category, 50 of the TLS connections were terminated with an `unexpected_message` alert, 8 with a `decrypt_error` alert, 8 with `certificate_unknown`, and 4 terminated with a `protocol_version` alert.

The remaining 1,372 faulty signatures sent no alerts, suggesting that these errors may have occurred during transmission or collection. In the UCSD data, 16 connections sent no alert but had an unsuccessful handshake, suggesting the client also failed to validate the signature. We found these signatures were formatted as a valid TLS 1.2 RSA signature using

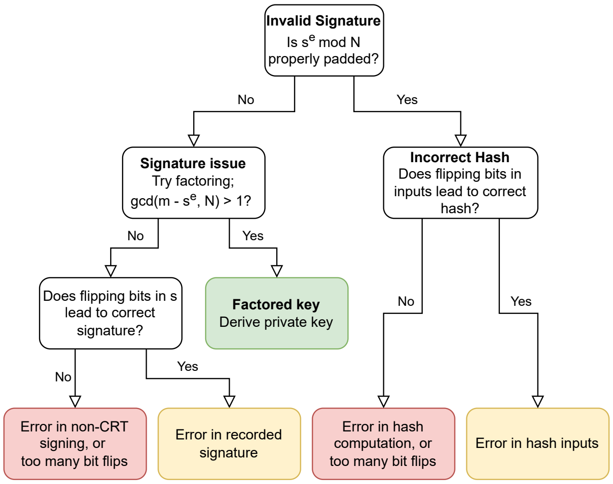
Invalid Signature
Is s^e mod N properly padded?
No
Yes
Signature issue
Try factoring;
gcd(m - s^e, N) > 1?
Incorrect Hash
Does flipping bits in inputs lead to correct hash?
No
Yes
Does flipping bits in s lead to correct signature?
Factored key
Derive private key
No
Yes
No
Yes
Error in non-CRT signing, or too many bit flips
Error in recorded signature
Error in hash computation, or too many bit flips
Error in hash inputs

Figure 2: **Factoring flow chart** — There are several potential causes for an observed invalid signature. If an error occurred during *transmission*, then we may be able to brute force bit flips in the signed values or signature to recover the original values. Alternatively, an error that occured during signature *computation* may allow us to factor the RSA key (green).

SHA256, but the connection used TLS 1.0, which computes the digest differently. These connections were between the same client and server over the period of a couple seconds.

In order to see if we could recover hypothesized bit flips during transmission, we brute forced both single-byte errors in every byte of the hash inputs and double bit errors anywhere in the hash inputs for all of the failed signatures to see if they would verify. None did. This suggests to us either that for the other invalid signatures there were multiple errors in the values, or that the hash function computation on the server side experienced a different type of failure.

Incorrect metadata For five signatures we recorded `0x00e6` for their signature algorithm extension, while for one we recorded `0x00d6`. Neither of these are valid values for this extension. These signatures did not seem to be formatted as a recognizable RSA padding scheme when raised to the public exponent and could not be verified. The clients and servers did not raise errors and did not complete these handshakes, so we do not know where the error occurred.

Faulty signatures Eliminating the two categories above left 200 signatures identified as RSA PKCS#1v1.5 in the handshake where the recovered faulty padded "message" m_f did not have a proper padding format.

For these signatures, we calculated the "correct" m expected from the collected handshake parameters and checked whether $\gcd(m - m_f, N)$ resulted in a factor of N. For 11 signatures, it did, allowing us to derive the corresponding RSA private key. We discuss these signatures and keys in more detail below in Section 4.1.2.

Among the remaining 189 signatures, 10 of the clients raised `decrypt_error` and did not complete the handshake. Of the 179 connections that did not raise an alert, 145 of them successfully completed the handshake.

For these signatures, we investigated whether a bit error during the transmission of the signature s might have resulted in the validation failure. We attempted to test this hypothesis by brute forcing all possible single-byte errors in s, but none resulted in a valid PKCS#1v1.5 padding format.

4.1.1 Investigating the CRT Errors

In each of the 11 connections in which we factored the RSA public key, the connection was made using TLSv1.2, so we were able to passively collect the signature and the inputs to the hash function used for signing. Three of the signatures used SHA-256 for the hash algorithm and eight used SHA-512 as the hash algorithm. Nine of the connections resulted in a TLS alert—eight `decrypt_error` and one `close_notify`—and the connection was unsuccessful in all eleven cases.

Nine of these faulty signatures were signed with (and revealed the factorization of) the same RSA modulus, and the two remaining ones revealed two further distinct RSA moduli.

Multiple faulty signatures from the same modulus gave us some opportunity to investigate the nature of the fault further. The GCD calculation revealed the same factor p each time, so we can conclude that the error occurred in the computation modulo the other factor, q, each time. We attempted to reproduce the error with single byte errors in combinations of the inputs to the CRT calculation: q, d_q, and q_{inv}. None of these was successful. We also computed $\gcd(m_i - s_i^e \bmod N, m_j - s_j^e \bmod N) = d_{ij}$ to see if we might recover information about an incorrect second factor, but each of these gave us p, $2p$, or $3p$.

4.1.2 baidu.com case study

The three private keys revealed by the 11 faulty signatures in our data were associated with three certificates that were served from four different IP addresses associated with Baidu. Two of the certificates were for `baidu.com`, and the third was for `httpsdns.baidu.com`. We confirmed these are the trusted CA-signed certificates served by Baidu services.

Figure 3 shows the number of connections made to these IP addresses during our collection period, with markers indicating days we observed the faulty signatures.

However, those certificates were not just used by the four IP addresses from which we observed faulty signatures. In our dataset these three certificates were used to secure 1.2 million distinct TLS connections, which were directed to 1,121 distinct IP addresses. There were 599 different values for the server name indication (SNI) among these connections.

In total, certificates with baidu.com in the common name field were associated with 37 failed signatures in our data, 30 of which ended with a `decrypt_error`, three without any error, two with `close_notify`, and one with `certificate_unknown`. Eleven of these signatures used PKCS#1v1.5 padding, and the remaining used RSA-PSS.

Server/code responsible After we disclosed to Baidu, they informed us that the traffic we observed was between the clients and Baidu's golang-based L7 load balancer BFE [4] which offloads cryptographic operations like signature generation to a hardware accelerator. Baidu did not share details of the specific cryptographic accelerators other than that Baidu's strategy was similar to other giant L7 load balancing deployments. There are numerous publicly available hardware implementations for such cryptographic workloads including Mellanox SmartNIC, DPU, Intel QAT, as well as FPGA and ASIC devices.

To fix the vulnerability, Baidu added a signature validation check in software. Based on the temporal pattern of signature errors we observed, we hypothesize that the errors may have been due to a single failing hardware component which then passed vulnerable signatures through the unprotected software implementation. We note that this failure mode is similar to the 2015 hardware failures described by Weimer [53] who traced RSA signature errors to a family of Cavium cryptographic hardware accelerators that were used, among other products, on TLS-terminating load balancers.

In order to investigate whether Baidu's implementation might be used elsewhere, we fingerprinted the Baidu server serving the vulnerable key based on its supported cipher suites and preference order [20] and observed that this fingerprint appeared to be unique to Baidu.

Our fingerprinting algorithm sends a TLS connection to the server claiming to support all known cipher suites, and observes which one the server selected. We then send a new TLS connection with the same list, removing the cipher suite the server chose. We repeat this process until the server sends an error (meaning they do not support any more cipher suites), and record the cipher suites and order they picked them. Baidu supported 7 cipher suites.

We then repeated this process to obtain server fingerprints for a 1% sample of TLS hosts. We used ZMap [21] to scan port 443, producing a list of 550K open hosts. Of these, 353K produced fingerprints without error, with 59K unique fingerprints. Only 38 of these matched the fingerprint of Baidu. Each of the 38 IPs presented a certificate for baidu.com.

We also experimented with the RSA private key fingerprinting technique of Janovsky et al. [33] to see if we could determine the implementation that generated the key. For each of the two keys we factored, we generated two candidate private keys, one for each ordering of prime factors p and q. We then ran Janovsky et al.'s fingerprinting tool on both private keys. For one of the orderings for each key the tool identified it as likely coming from OpenSSL (their group 24) with 90% confidence. For the other prime ordering, the tool suggested other libraries with lower confidence.

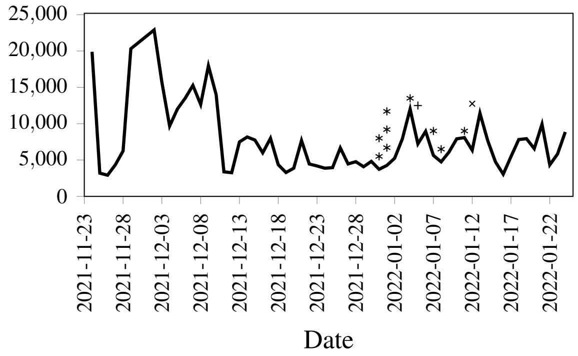

Figure 3: **Total observed connections over time to the four IP addresses that generated a factorable signature.** The markers indicate when the connections with faulty signatures occurred. Different shaped markers represent different moduli.

Active Analysis of baidu.com Servers After discovering the errors in January 2022, we initiated TLS connections to the four Baidu IP addresses associated with the faulty signatures in an attempt to reproduce a faulty signature in a controlled environment. We completed 500,000 connections from two IP addresses, with randomized TLS client hello fingerprints using uTLS [24].

Of these 500,000 completed connections, one signature did not verify, and our client produced a TLS alert. The signature used the rsa_pss_rsae_sha512 signature algorithm, so the invalid signature did not allow us to extract the factorization of the public key.

We made an effort to mitigate negative effects on server performance by rate limiting our scan so that our connections would be expected to be only a negligible proportion of traffic for this popular site.

4.2 PKCS#1v1.5 Implementation analysis

To see whether there are open source libraries vulnerable to key compromise via signature faults, we manually reviewed source code for a sample of popular open source TLS libraries. All of the libraries we looked at had code to validate signatures before returning, specifically aimed at preventing this vulnerability. All of the libraries also used RSA blinding and the Chinese Remainder Thereom when producing signatures. Most recent versions also validate the produced signatures.

Table 5 lists the libraries we looked at, and the version/date that signature validation checks were added to the codebase.

[4] https://github.com/bfenetworks/bfe

Versions of libraries that predate these are likely vulnerable.

Library	Version fixed	Date of release
OpenSSL	0.9.6a	Mar. 2001
Golang TLS	1.6	Dec. 2015
OpenJDK	jdk8u	Feb. 2015
Mbed TLS	2.1.0	Sep. 2015
libgcrypt	1.7.0	Aug. 2015
wolfSSL	3.6.8	Sep. 2015
wolfCrypt	4.3.0	Nov. 2019
Nettle	3.2	Sep. 2015

Table 5: **TLS implementations**— All of the open source TLS libraries we examined have implemented fixes that correct this issue, most commonly by verifying the signature before returning. We note that some of these checks can be disabled by application configurations.

Some libraries, including OpenJDK, have a configurable signature check, which applications can disable for performance reasons. We cannot tell a priori if applications will provide their own configurations when using this library and make them vulnerable. Similarly, wolfSSL added signature verification for its TLS code in 2015, but the signature check was not incorporated into wolfCrypt, a lower-level cryptographic library, until several years later. Applications that used wolfCrypt to produce RSA signatures remained vulnerable until a patch was added in 2019 in response to a Rowhammer-based fault attack [54].

Finally, OpenSSL's signature check is unique among the libraries we examined. Rather than simply clear the signature or return an error if the signature fails, OpenSSL recomputes the signature *without* using the CRT optimization (i.e. computes $m^d \bmod N$ directly) when its first attempt fails. This "second try" signature is not checked before it is returned to the user. As discussed in Section 2.2, there are algorithmic fault attacks that don't rely on CRT computations, but they seem more challenging to exploit.

4.3 RSA-PSS

Of the non-validating signatures in our sample, 78 were signed with RSA using PSS padding. Of these, 25 terminated with the TLS alert `decrypt_error`, 13 with `decode error`, 2 with `close_notify`, and the remaining 38 did not raise an alert. Of these 38, half of them successfully completed the handshake.

RSA-PSS padding contains a fixed flag byte `0xBC` that should be present in the recovered padded message $m = s^e \bmod N$ derived from a signature s. For each of the faulty PSS signatures, we checked for the presence of the flag byte and found that it was present in 30 connections.

For connections missing the flag byte this suggests that the faults were either due to errors in the modular exponentiation in the signing process, or to bit errors in the signature value during transmission. For the connections with the flag byte this suggests that the error occurred in the computation of the hash.

To investigate the possibility of bit errors during transmission, we brute forced all possible single-byte errors in the signature and did not find any that resulted in validating signatures. To check for errors in the hash generation we brute forced single-byte errors in the hash inputs and also failed to find any that resulted in matching hashes.

RSA-PSS has been proven secure against fault attacks if the salt is randomized and unpredictable [18]. However, if the salt is predictable and the padded message hash to be signed is known to the attacker, then PSS would become vulnerable to the same CRT-based fault attack as any other deterministic RSA padding.

4.3.1 Is RSA-PSS randomized in practice?

We were intrigued by the note in PKCS#1v2.1, discussed in Section 2.1, suggesting that randomization is not critical to the security of PSS and therefore that implementations may use fixed values or sequence numbers as a salt value.

We examined the salt values used to generate the valid PSS signatures in our dataset in order to verify whether the signatures were in fact randomized. This is straightforward: given a signature s, we recover $m = s^e \bmod N$, verify the `0xBC` flag byte on m, then parse m into `maskedDB` and `H`, and recover the padded salt by xoring `maskedDB` with `H`.

We found repeated salt values in 148 out of 330 million PSS signatures, suggesting that repeats exist, but seem to be rare enough to make a practical transient fault attack unlikely outside of more specific implementation choices.

4.3.2 Key reuse between PKCS#1v1.5 and RSA-PSS

It was quite common in our dataset to find both PKCS#1v1.5 and RSA-PSS signatures from the same public key. In the CU Boulder dataset, of 498,575 distinct RSA public keys observed, 33,320 of them had generated both PKCS#1v1.5 and RSA-PSS signatures. These keys made up 37% of the 1.6 billion total TLS 1.2 connections seen on this network. In the UCSD dataset we observed 516,019 distinct RSA keys, of which 21,916 had generated both PKCS#v1.5 and RSA-PSS signatures, accounting for approximately 34% of the 1.7 billion TLS 1.2 connections.

Kakvi [36] has formally proven that this type of key reuse is secure, with a focus on the context of TLS 1.3 supporting both. However, this proof naturally does not include fault attacks, since they fall outside of the normal UF-CMA security model for digital signatures; from that perspective, supporting multiple padding schemes creates a larger attack surface for a given private key. TLS has historically used the same long-term private keys for different protocol versions, both encryption and signatures, and different cipher and algorithm options, which has enabled a variety of downgrade attacks [1, 3, 7].

4.4 ECDSA statistics

In the UCSD data, we observed 121 million ECDSA signatures, and at CU Boulder we observed 77 million. At UCSD all but 260 of these signatures validated, while CU Boulder identified 207 signatures that failed to verify.

We investigated whether the circumstances necessary for a successful passive fault ECDSA attack might arise in the data we observed. Recall from Section 2.4 that a two-signature ECDSA fault attack requires a correct and a faulty signature using the same message hash h and signature nonce k. For an implementation using deterministic ECDSA nonce generation, we would expect to see the same nonce k used every time a message hash h is signed.

However, in the TLS context, it seems less plausible that a passive network attacker might see multiple signatures over the same message hash h, since the server signature includes the client and server randoms, which are supposed to be freshly randomly generated for each handshake. Thus in principle, TLS ECDSA signatures should not be vulnerable to fault attacks.

Connections w/ repeated	UCSD	CU Boulder
Server Random	58K	136
Client Random	77K	25K
Server Parameters	191M	176M
Client Random and Server Random	29K	0
Client Random and Server Parameters	919	839
Server Random and Server Parameters	1.1K	0
Client Random, Server Random, and Server Parameters	904	0
Total Connections Seen	1.3B	1.4B

Table 6: **Repeated nonces and key exchanges in TLS 1.2.** We find that non-unique ECDHE parameters remain common among servers, and a small fraction of clients and servers use non-unique randoms.

In practice, however, we observed a small but non-negligible number of handshakes that repeated all of these values. Table 6 summarizes the repeated values across our observed connections.

Repeating client and server nonces In the UCSD data we observed 6,566 distinct client random values that appeared in more than one connection. 77,354 connections used one of these non-unique values. 1,936 distinct server random values appeared in more than one connection; these appeared in 58,099 connections. Of these connections, 1,210 of the non-unique client randoms and 308 of the non-unique server randoms appeared in connections that included ECDSA signatures. Furthermore, 74 of these connections with ECDSA signatures repeated not only the client and server random, but also the Diffie-Hellman key exchange parameters; there were 37 distinct duplicated sets of values.

Repeating server key exchange parameters Although (elliptic curve) Diffie-Hellman is often described as providing "perfect forward secrecy" because in principle both the client and server generate fresh key exchanges for every connection, this is not actually the case for many implemenations. In particular, many servers have been observed to reuse Diffie-Hellman key exchange parameters for days to months [51].

In OpenSSL, this behavior was historically governed by the `SSL_OP_SINGLE_DH_USE` flag; as of OpenSSL version 1.0.2f released in 2016, this option is always on.

Our data shows that non-unique server ECDH parameters are very common; in the UCSD data almost 15% of observed connections used a non-unique set of server key exchange parameters.

4.4.1 Compromised ECDSA keys from insecure RNGs

Although it is not the main focus of our paper, we also checked our collected ECDSA signatures for repeated signature nonces that would allow us to compute a server's private signing key.

In our data we found 637 connections with 183 different repeated signature r values, allowing us to compute two different ECDSA private keys. All the connections were to the same IP address, which we identified as a Xerox AltaLink C8155 printer located on the UCSD campus. This server was repeating its server random value, its chosen Diffie-Hellman parameters, and its choice of k on new connections with the same client. In our data the repeated values were close in time with a maximum interval of five minutes in between connections. The values did not appear to be hard-coded, since they differed across different connections over time. In total, we saw 7,043 connections to this server all using one of the two compromised keys. We hypothesize that a faulty random number generator led to this vulnerability.

5 Active scanning

To supplement the passive network data, we also analyzed historical scan data and performed more recent active scans of TLS.

In total, we derived 127 distinct private keys from 128 hosts with faulty signatures across all our active scans. These include 54 keys revealed from historical scans between 2015 and 2020, and 73 keys computed from three weeks of daily active scans in 2022.

5.1 Collecting TLS signatures from old scans

The historical scan data we collected is a patchwork of full IPv4 scans carried out using ZMap and ZGrab, beginning in 2015. The majority of the data was originally downloaded from `scans.io`. This regular data collection was taken over by Censys [19] who now carries out regular internet-wide scans, but they do not collect the TLS handshake data we

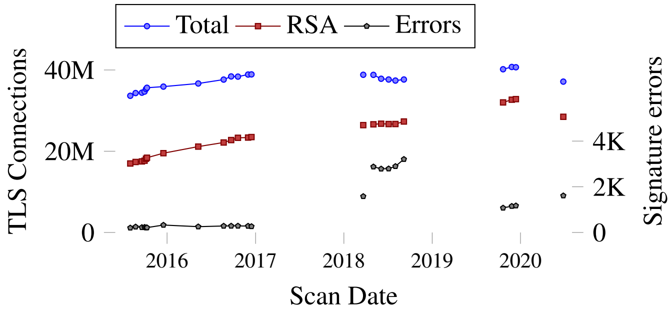

Figure 4: **RSA signatures and errors over time.** We examined historical full IPv4 TLS scans on port 443 that were collected using ZMap/ZGrab between 2015 and 2020 for different research projects. Although some of the scans were carried out using different methodology and may have requested different cipher suites, we see that the number of successful connections, the number of connections that included an RSA signature, and the number of RSA signatures that did not validate have all risen over time.

need for our analysis. Since 2020, their dataset no longer includes TLS signatures, and it appears that the Censys Universal Internet Dataset generated from scan data prior to the deprecation of handshake signatures in 2020 only includes successful handshakes, and thus omits the errors we are looking for. We analyzed 2019 ZGrab scan data provided as a paper artifact by Wan et al. [52], and a 2020 TLS scan provided by Izhikevich [31].

Because the historical scans we analyze were collected for a variety of previous research projects, the scan methodology and information collected has changed over the years, and there are gaps in the data.

Figure 4 shows the number of successful TLS handshakes for these scans over time, the number of handshakes containing RSA signatures, and the number of RSA signatures that did not validate in each scan.

Most significantly for our analysis, ZGrab's TLS module does not record the signature hash. For scan data through 2015, the records contain all of the handshake values necessary to reconstruct the hash (client and server randoms and server key exchange). However, after 2015 ZGrab stopped recording the client random from the exchange and does not save the signature hash.

For later scans, we were able to use the approach of Coron et al [17] to identify vulnerable signatures for some choices of hash function and key length. In particular, for 1024-bit RSA, we were only able to recover keys from faulty signatures using SHA-1, because longer hash lengths were prohibitively slow (SHA256) or infeasible (longer hash lengths). For 2048-bit keys, we could recover keys from faulty signatures using SHA1, the 36-byte combination of MD5 and SHA1 in TLS 1.0 and 1.1, and SHA256. For larger keys, key recovery was efficient for any hash function.

5.2 Current scan data

In order to get a clearer picture of the current internet from a scanning perspective, we also carried out three weeks of daily TLS scans focused on finding keys from invalid signatures. The scans were carried out from a dedicated scanning machine located at UCSD during May and June 2022.

Each daily scan was carried out in two steps. We first scanned the entire IPv4 address space on port 443 using ZMap [21] to identify hosts that completed a TCP handshake. These scans gave 53.0–53.4 million IP addresses responding on port 443. We then made rate-limited TLS handshakes to each responsive IP address with a version of ZGrab2 modified to save the signature hash value. We performed scans with two lists of cipher suites: first using ZGrab2's deafult list (some of which do not involve a signature for key exchange), and second using a list that only included DHE and ECDHE cipher suites that required signatures from the server. Our scans using the default cipher suite list resulted in around 30 million RSA signatures (out of 35 million TLS connections), of which 850 signatures were invalid on average per scan. Our second list resulted in 32.7 million RSA signatures (from 32.8 million connections), with around 1,800 invalid signatures per scan. The scans cover TLS 1.0–1.2, but do not include TLS 1.3 handshakes, which ZGrab2 does not support. We note that TLS 1.3 uses RSA-PSS, which has been proven secure against fault attacks for properly-randomized salts [18], so we would not expect to find faulty signatures that allow key recovery in TLS 1.3 (see Section 4.3).

5.3 Historical and current scan results

From our scan data, there appear to be multiple classes of errors. Some hosts reappear with compromised signatures in scans over the course of months to years. This would suggest that the errors they experience are persistent: disk corruption or memory corruption affecting the private key. For some of these hosts that are still online, we were unable to complete a TLS handshake with them due to persistent handshake errors.

In other cases, the signature errors appear to be transient: 56 of the keys we find appear only once across all our scans.

5.3.1 Affected devices

The landscape of vulnerable keys has changed since 2015, but it remains the case that most of the vulnerable keys are associated with network devices rather than general-purpose trusted servers. Our 2015 scan data reflects a very similar picture to Weimer's report: we recovered keys associated with Hillstone Networks, Viprinet, QNO, and Fortinet. Some of these devices are quite long lived: we see a QNO device in scans as late as 2019. We also see certificates from devices

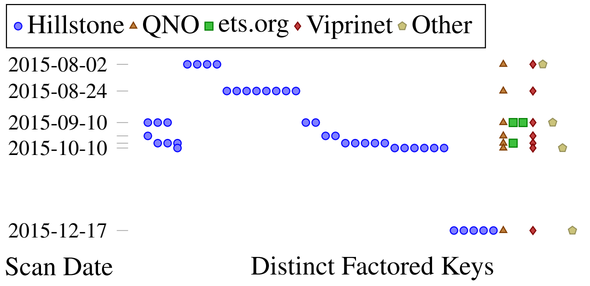

Figure 5: **Persistence of factored keys in 2015 scans.** In our examination of historical scan data from 2015, we clustered factored keys by the device or hostname identified by the certificate. We see some hosts with persistent errors in every scan. For hosts whose certificates identified Hillstone devices, the certificates all had the same common name, but the factored keys seen in each scan were generally distinct keys served from distinct hosts, suggesting a more sporadic error pattern across a larger number of devices.

that weren't mentioned by Weimer, including SonicWALL. None of these certificates were CA-signed or browser-trusted.

Figure 5 shows the distribution of keys associated with different types of devices or other host identifiers in our historical 2015 scan data over time. The 2015 scan data also includes semi-persistent errors on hosts with certificates under `ets.org`, which no longer appear after October 6, 2015. The 2019 scan data reveals private keys for a handful of sites with browser-trusted certificates, including a subdomain of `doi.gov` which is no longer accessible to the public.

In our 2022 scan data, nearly all of the compromised keys that we were able to identify were associated with a variety of Cisco VPN products. However, in contrast to the situation in 2015, twenty-nine of these domains served browser-trusted certificates, including nine wildcard certificates. Of these 29 certificates, 20 were still valid, 6 had expired, and 3 had been revoked. Many of the non-wildcard, browser-trusted certificates identified a subdomain for a VPN. Many of these hosts had persistent signature errors and appeared in most or all of our daily scans. Figure 6 shows the distribution of factored keys that appeared in our daily scans from 2022.

In addition to these device-associated keys, we also observed sporadic errors from a small handful of hosts that appeared to be normal web servers running on Alibaba Cloud that were running Microsoft IIS 7.0 on Windows Server 2008. The RSA private key fingerprinting tool of Janovsky et al. [33] did not identify their private keys as having been generated by a family of implementations that includes Microsoft CryptoAPI or Microsoft CNG. We performed follow up scans focused on Alibaba Cloud prefixes. From 26 scans of Alibaba Cloud, we discovered 8 keys from faulty signatures.

From discussions with Alibaba and Microsoft, it appears that the Microsoft TLS implementation separates the protocol layer from the implementations of cryptographic primitives, and provides a cryptographic API for smart cards, TPMs, or other implementations. The software KeyStorageProvider (KSP) that ships with Windows performs signature validation, but the TLS implementation may not validate in some cases. Thus a server that offloads cryptographic operations to an HSM or hardware accelerator that does not perform validation may potentially be vulnerable to failing hardware.

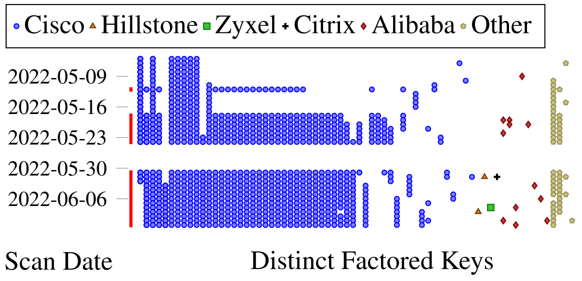

Figure 6: **Persistence of factored keys in 2022 scans.** For many of the hosts producing signature errors in our scans, the errors appear to be persistent across nearly every scan. Other hosts seem to produce only sporadic errors. The red line on the y axis marks scans that requested only (EC)DHE cipher suites, which collects more faulty signatures.

6 Defenses

Our results demonstrate yet again that RSA PKCS#1v1.5 and other deterministic digital signature schemes should be considered *fragile*, and that any implementation of deterministic RSA signature padding or ECDSA nonce generation should include an extra signature validation step to protect against transient faults.

Validate signatures before sending Any implementation using RSA-CRT or deterministic ECDSA nonce generation to compute signatures *must* validate all signatures before sending them across the wire. As discussed in Section 4.2, nearly every open-source RSA implementation we examined already does this.

Randomization Signature randomization appears to provide a real benefit against fault attacks. With ECDSA, however, signature randomization is a double-edged sword: any randomness failure for randomly generated signature nonces results in a compromise of the long-term key. This tension has arisen in standards discussions; see for example a discussion that fortuitously arose as we were writing this paper [22]. A middle ground that should provide protection against both random number generator vulnerabilities and signature faults is to generate an ECDSA signature nonce from the private key, message, and a random input [44].

Avoiding CRT computations The RSA vulnerability we exploit is entirely due to the use of CRT computations. If an implementation computes an RSA signature by carrying out a modular exponentiation with the full secret exponent d, then a fault will not allow an attacker to recover the factorization of the key in the same way. This countermeasure is already in use by OpenSSL: if the RSA signature validation fails, the signature is recomputed using only the private key d.

As noted in Section 2.2, fault attacks against non-RSA-CRT exponentiation with d can still be vulnerable to fault attacks, although these attacks seem less plausible to exploit via spontaneous hardware errors.

Non-defenses Blinding, either of signature or exponent, is not a defense against these types of fault attacks. The only thing the RSA attacks require is for any type of fault to occur in one of the signature components computed for the RSA-CRT reconstruction. A fault could still occur if the signature is computed blinded.

7 Discussion

7.1 The persistence of PKCS#1v1.5 padding

PKCS#1v1.5 padding for both RSA encryption and signatures has had a surprisingly long life despite the repeated demonstrations of catastrophic attacks breaking the security of both. In both cases, it appears that this has been due to backwards compatibility concerns. TLS 1.3 has removed both RSA encryption and PKCS#1v1.5 signature padding, replacing them with (EC)DH key exchange and RSA-PSS, ECDSA, or EdDSA, respectively. Both of these changes appear to have caused compatibility issues that are slowing the adoption of TLS 1.3 [6, 47]. Both PKCS#1v1.5 and PSS padding are approved in the most recent FIPS 186-5 draft Digital Signature Standard (DSS) [41].

7.2 Side Channel and Fault Threat Models

Most of the academic work on fault attacks in the context of RSA has been carried out under an active attack model, and published in the side channel literature. Protecting implementations against every variant of side channel attacks is currently impossible, and OpenSSL has publicly declared that "Certain threats are currently considered outside of the scope of the OpenSSL threat model. Accordingly, we do not consider OpenSSL secure against the following classes of attacks: same physical system side channel; CPU/hardware flaws; physical fault injection; physical observation side channels (e.g. power consumption, EM emissions, etc)."

While OpenSSL *has* implemented defenses against RSA signature faults, we hope that our observations illustrate that faults in computations do not require a physical attacker, fault injection, or a co-located process carrying out a Rowhammer attack: they may require only a failing hardware device.

7.3 Certificate risks

The patterns of compromise we observe illustrate the risk that faulty hardware can pose to long-term secret keys. In particular, if an organization uses a wildcard certificate with a VPN implementation that leaks that key, an attacker could reuse the private key to impersonate or decrypt traffic to other hosts protected by that certificate. This illustrates the importance of domain separation.

7.4 Re-examining the past

While our passive attack demonstrates lessons that are important for the future, it is equally relevant to re-examine this attack from a historical context. As discussed in [53] and in Section 4.2, prior to 2015 nearly all popular TLS libraries were vulnerable to RSA key recovery via signature faults. Given this historical implementation environment, it is natural to wonder if this attack could have been exploited in secret.

The passive nature of an RSA-CRT fault attack makes it an especially appealing avenue for a nation-state who is able to perform large-scale network surveillance and has an incentive to decrypt TLS traffic at scale: they can avoid sending large numbers of TLS connections to vulnerable sites, which would scale poorly and leave evidence in observant websites' logs. Instead, the adversary only needs to passively observe large amounts of network traffic, a capability that many nation-states are known to have, and wait for a transient error. For instance, the United States' NSA is known to have deployed global network taps to collect and analyze large amounts of Internet traffic prior to 2015 [27].

The state of common TLS cipher suites prior to 2015 paints an even bleaker picture. Until a few years ago, non-forward secure RSA key exchanges dominated TLS connections: according to a leaked classified NSA document on SSL trends from the Communications Security Establishment Canada, over 95% of TLS connections used an RSA key exchange in 2011 [50]. Since TLS versions prior to 1.3 reuse the RSA keys in a server's certificate both for digital signatures (if client and server negotiate a Diffie-Hellman cipher suite) or for encryption (if client and server negotiate RSA key exchange), an attacker could use signature data to passively recover the RSA private key, and then use this private key to decrypt observed RSA-encrypted data from other past or future connections.

7.5 Prospects for the future

The rise of TLS 1.3 spells a slow end to access to the trove of metadata exposed to passive network observers by the handshakes for TLS versions 1.2 and below. In addition, TLS

1.3's removal of a number of insecure cipher options will eventually lead to the end of families of cryptographic attacks that have plagued TLS security for decades.

The adoption of TLS 1.3 does not, however, entirely remove the possibility of the attacks we discuss. TLS certificates are typically shared across multiple versions of SSL/TLS supported by a server, and as long as support for TLS 1.2 and below exists, an attacker could potentially carry out a TLS 1.3 impersonation or man-in-the-middle attack against an unprotected implementation that inadvertently exposes its private key through a faulty TLS 1.2 signature.

However, on the bright side, even for these earlier TLS versions, the security of the cipher suites offered by servers and negotiated by clients has improved dramatically over the past decade, thanks to increased attention, and all signs point to the continuing ubiquity of TLS and encrypted protocols and improvements in cryptographic security for all [48].

8 Acknowledgements

We thank Zakir Durumeric and Liz Izhikevich for sharing their 2019 and 2020 active scan data, Steve Weis, Asil Veral, Yu Ding, Andrew Chi, David McGrew, and Dan Shumow for their help with the disclosure process, and the anonymous reviewers for their constructive suggestions. This research was supported by NSF grants no. 2048563 and 2145783, and the DARPA RACE program.

References

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 5–17, Denver, CO, USA, October 12–16, 2015. ACM Press.

[2] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 260–275, Redwood Shores, CA, USA, August 13–15, 2003. Springer, Heidelberg, Germany.

[3] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 689–706, Austin, TX, USA, August 10–12, 2016. USENIX Association.

[4] Michael Barr. Bookout v. Toyota 2205 Camry L4 software analysis. https://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf, 2013.

[5] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 399–416, Saragossa, Spain, May 12–16, 1996. Springer, Heidelberg, Germany.

[6] David Benjamin. TLS ecosystem woes: Why your crypto isn't real world yet. Talk at Real World Crypto 2018.

[7] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[8] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 1–12, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[9] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. A new CRT-RSA algorithm secure against Bellcore attacks. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *ACM CCS 2003*, pages 311–320, Washington, DC, USA, October 27–30, 2003. ACM Press.

[10] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of bleichenbacher's oracle threat (ROBOT). In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 817–849, Baltimore, MD, USA, August 15–17, 2018. USENIX Association.

[11] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 37–51, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany.

[12] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):101–119, March 2001.

[13] Dan Boneh et al. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[14] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 157–175, Christ Church, Barbados, March 3–7, 2014. Springer, Heidelberg, Germany.

[15] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 3–20, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019. Springer, Heidelberg, Germany.

[16] Eric Brier, David Naccache, Phong Q. Nguyen, and Mehdi Tibouchi. Modulus fault attacks against RSA-CRT signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 192–206, Nara, Japan, September 28 – October 1, 2011. Springer, Heidelberg, Germany.

[17] Jean-Sébastien Coron, Antoine Joux, Ilya Kizhvatov, David Naccache, and Pascal Paillier. Fault attacks on RSA signatures with partially unknown messages. In Christophe Clavier and Kris Gaj, editors, *CHES 2009*, volume 5747 of *LNCS*, pages 444–456, Lausanne, Switzerland, September 6–9, 2009. Springer, Heidelberg, Germany.

[18] Jean-Sébastien Coron and Avradip Mandal. PSS is secure against random fault attacks. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 653–666, Tokyo, Japan, December 6–10, 2009. Springer, Heidelberg, Germany.

[19] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by internet-wide scanning. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 542–553, Denver, CO, USA, October 12–16, 2015. ACM Press.

[20] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. The security impact of https interception. In *Network and Distributed System Symposium*, 2017.

[21] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast internet-wide scanning and its security applications. In *In Proceedings of the 22nd USENIX Security Symposium*, 2013.

[22] Stephen Farrell. EDDSA (un)suited for mandatory to implement ciphersuite? https://mailarchive.ietf.org/arch/msg/cfrg/Ev8hgyojKeObXMZ7SF2m3_yekMo/, 2022.

[23] Apache Software Foundation. Apache Hive website. https://hive.apache.org/, 2019.

[24] Sergey Frolov. uTLS. https://github.com/refraction-networking/utls.

[25] Sergey Frolov and Eric Wustrow. The use of TLS in censorship circumvention. In *Network and Distributed System Symposium*, 2019.

[26] Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, 1998.

[27] Glenn Greenwald. XKeyscore: NSA tool collects ‘nearly everything a user does on the internet’. *The Guardian*, 2013.

[28] Peter Gutmann. Why RSA-PSS is much less secure than PKCS #1 v1.5. https://www.metzdowd.com/pipermail/cryptography/2019-November/035449.html, 2019.

[29] Marcella Hastings, Joshua Fried, and Nadia Heninger. Weak keys remain widespread in network devices. In *Proceedings of the 2016 Internet Measurement Conference*, IMC ’16, page 49–63, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security 2012*, pages 205–220, Bellevue, WA, USA, August 8–10, 2012. USENIX Association.

[31] Liz Izhikevich, Renata Teixeira, and Zakir Durumeric. LZR: Identifying unexpected internet services. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 3111–3128. USENIX Association, August 11–13, 2021.

[32] Tibor Jager, Saqib A. Kakvi, and Alexander May. On the security of the PKCS#1 v1.5 signature scheme. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1195–1208, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[33] Adam Janovsky, Matús Nemec, Petr Svenda, Peter Sekan, and Vashek Matyas. Biased RSA private keys: Origin attribution of GCD-factorable keys. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*,

pages 505–524, Guildford, UK, September 14–18, 2020. Springer, Heidelberg, Germany.

[34] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1:36–63, 2001.

[35] J. Jonsson and B.Kaliski. Public-key cryptography standards (PKCS) #1: RSA cryptography specifications version 2.1. Technical report, Internet Society, 2003.

[36] Saqib A. Kakvi. On the security of RSA-PSS in the wild. In *Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop*, SSR'19, page 23–34, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Burt Kaliski. PKCS# 1: RSA encryption version 1.5, 1998.

[38] Chong Hee Kim and Jean-Jacques Quisquater. Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures. In *IFIP International Workshop on Information Security Theory and Practices*, pages 215–228. Springer, 2007.

[39] Arjen K Lenstra. Memo on RSA signature generation in the presence of faults. Technical report, EPFL, 1996. https://infoscience.epfl.ch/record/164524.

[40] ntop. PF_RING: High-speed packet capture, filtering and analysis. https://www.ntop.org/products/packet-capture/pf_ring/.

[41] National Institute of Standards and Technology. FIPS 186-5: Digital signature standard (dss). https://csrc.nist.gov/publications/detail/fips/186/5/draft.

[42] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Netw.*, 31(23–24):2435–2463, December 1999.

[43] Andrea Pellegrini, Valeria Bertacco, and Todd Austin. Fault-based attack of RSA authentication. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 855–860. IEEE, 2010.

[44] Trevor Perrin. The XEdDSA and VXEdDSA signature schemes. https://signal.org/docs/specifications/xeddsa/#security-considerations, 2016.

[45] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 338–352, 2018.

[46] T. Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). Technical report, RFC 6979, August, 2013. https://datatracker.ietf.org/doc/html/rfc6979.

[47] Thomas Pornin. BearSSL TLS 1.3 status. https://www.bearssl.org/tls13.html.

[48] Qualys. SSL pulse. https://www.ssllabs.com/ssl-pulse/.

[49] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1–18, Austin, TX, USA, August 10–12, 2016. USENIX Association.

[50] (redacted) Communications Security Establishment Canada. TLS trends: A roundtable discussion on current usage and future directions. 2012. https://www.spiegel.de/media/25722dbd-0001-0014-0000-000000035510/media-35510.pdf.

[51] Drew Springall, Zakir Durumeric, and J. Alex Halderman. Measuring the security harm of tls crypto shortcuts. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 33–47, New York, NY, USA, 2016. Association for Computing Machinery.

[52] Gerry Wan, Liz Izhikevich, David Adrian, Katsunari Yoshioka, Ralph Holz, Christian Rossow, and Zakir Durumeric. On the origin of scanning: The impact of location on internet-wide scans. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 662–679, New York, NY, USA, 2020. Association for Computing Machinery.

[53] Florian Weimer. Factoring RSA keys with TLS perfect forward secrecy. Technical report, Red Hat, 2015. https://www.redhat.com/en/blog/factoring-rsa-keys-tls-perfect-forward-secrecy.

[54] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on heterogeneous FPGA-CPU platforms. *IACR TCHES*, 2020(3):169–195, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8587.

[55] Junko Yoshida. Toyota case: Single bit flip that killed. *EE Times*, 2013. https://www.eetimes.com/toyota-case-single-bit-flip-that-killed/.