Everlasting ROBOT: the Marvin Attack

Hubert Kario$^{[0009-0007-8694-4270]}$

Red Hat Czech s.r.o, Purkyňova 115, 612 00 Brno, Czech Republic
hkario@redhat.com

16 May 2023

Abstract. In this paper we show that Bleichenbacher-style attacks on RSA decryption are not only still possible, but also that vulnerable implementations are common. We have successfully attacked multiple implementations using only timing of decryption operation and shown that many others are vulnerable. To perform the attack we used more statistically rigorous techniques like the sign test, Wilcoxon signed-rank test, and bootstrapping of median of pairwise differences. We publish a set of tools for testing libraries that perform RSA decryption against timing side-channel attacks, including one that can test arbitrary TLS servers with no need to write a test harnesses. Finally, we propose a set of workarounds that implementations can employ if they can't avoid the use of RSA.

Keywords: Side-channel attacks · timing attacks · Bleichenbacher attack · RSA.

1 Introduction

While the web traffic increasingly depends on the new ECDSA cryptosystem, majority of server certificates still use the RSA cryptosystem that was originally published in 1977. RSA saw first big use with the deployment of the Netscape Navigator 1.0 browser in 1994, as part of the SSL 2.0 protocol. Soon after that, in 1998, Daniel Bleichenbacher published a practical attack[2] on an SSL server due to both the faulty PKCS#1 v1.5 padding scheme and faults in the SSL protocol.

This was only the first of many attacks that followed. Large contributions to attacking RSA made by Manger[5] in 2001, Klíma, Pokorný, et al.[4] in 2003, Bardou et al.[1] in 2012, Meyer, Somorovsky, et al.[6] in 2014.

Despite those and other attacks being published, and an updated (also in 1998) version of the PKCS# 1 including the OAEP padding scheme, which is much more resistant against the attack published by Bleichenbacher, the vulnerable PKCS#1 v1.5 padding still remains in widespread use.

The SSL and TLS protocols never received an update to use the OAEP padding scheme for the RSA key exchange. It was only version 1.3 of the protocol, published in 2018, completely removed support for this key exchange[12].

Many other widely used cryptographic protocols, like S/MIME or JSON Web Tokens (JWT)[10], still allow use of PKCS#1 v1.5 padding for encryption.

Despite the original attack being nearly quarter of a century old, we found many commonly used implementations to be still vulnerable to it.

As it's a continuation of the ROBOT vulnerability[3], which we don't expect to get rid of any time soon, we've decided to name it after one everlasting "Paranoid Android".

1.1 Contributions

Our work makes the following contributions:

- We show that by using correct statistical methods and proper test setup, detection of side channels in RSA implementations is robust
- We show that multiple popular implementations, including ones previously tested, are still vulnerable to attacks utilising timing based oracles, both over loopback and over regular Ethernet networks.
- We publish a set of tools for testers of cryptographic libraries for checking the timing of APIs providing RSA decryption with minimal dependencies.
- For implementations which can't remove support for PKCS#1 v1.5 encryption, we propose an alternative decryption algorithm, which does not require side-channel free code on the application side.

2 Adaptive chosen ciphertext attacks

The Bleichenbacher attack allows decrypting arbitrary RSA ciphertexts or forging signatures when the attacker can learn some information about specially crafted ciphertexts, related to the ciphertext they want to decrypt[2].

The attack works thanks to few properties of the RSA cryptosystem:

- the RSA encryption is homomorphic with regards to multiplication,
- the PKCS#1 v1.5 padding requires specific values of the two most significant bytes: 0 and 2, but not for the whole padding,
- learning about PKCS#1 v1.5 conformance of a related ciphertext provides specific bounds on the value of the plaintext we want to decrypt.

Homomorphism in RSA cryptosystem Let e and n be the RSA public key (with e representing the public exponent and n representing the modulus), and let d be the corresponding secret key (the private exponent).

The RSA encryption operation of a message m is equal to $m^e \mod n$, giving ciphertext c. The RSA decryption operation of the ciphertext c is equal to $c^d \mod n$.

See the PKCS#1[11] specification for information on how n, d, and e are related to each-other, but it's not necessary to understand the attack.

Now, when we introduce some number s, then by calculating $c' = s^e c \mod n$ we effectively multiply the plaintext m by s, as $c' = s^e m^e = (sm)^e \mod n$. We can do that even when we don't know the value of m, just the value of it after encryption: c. Thus, we can multiply an arbitrary encrypted value by a value we know, just by having access to the public key.

PKCS#1 padding In the original attack[2], the attacker learns only whether or not a ciphertext decrypts to a correctly padded PKCS#1 v1.5 plaintext.

A plaintext is correctly padded when the number m converted to a big-endian representation of same size as the modulus n consists of 8-bit bytes as follows: 0x00, 0x02, PS, 0x00, P. Additionally the string PS consists of at least 8 bytes, none with value 0. The string P can include bytes of value 0 and can also be empty.

That means, that if a ciphertext is PKCS#1 conforming, we know that it decrypts to a value $2B \leq ms \mod n < 3B$, where $B = 2^{8(k-2)}$ and k is the number of bytes necessary to represent n as as a big-endian integer (i.e. it's larger or equal to 0x000200...00, but smaller than 0x000300..00).

Bleichenbacher attack Note that if we know that $c \cdot s^e \mod n$ is PKCS #1 conforming, it means that $c \cdot s \mod n \in [2B, 3B)$. That implies that there is an integer r such that $2B \leq m \cdot s - r \cdot n < 3B$. Equivalently:

$$\frac{2B + rn}{s} \leq m < \frac{3B + rn}{s} \tag{1}$$

By finding multiple s numbers, we find multiple r values, which provide more and more restrictive ranges of m.

See the original analysis[2] for some optimisations on how to find values of s that are more likely to create PKCS conforming ciphertexts.

While the original paper required about one million calls to the oracle (checks if a message is PKCS#1 conforming or not), more recent analysis performed by Bardou et al.[1] shows that as few as 3800 oracle queries may be enough to decrypt a 1024 bit message.

RSA OAEP encryption PKCS#1 version 2.0[7] specified a different padding format for RSA encryption intended to defeat attacks like the one proposed by Bleichenbacher.

While the standard specifies that the decryption needs to ignore the value of the most significant byte of plaintext, some implementations do not do that. This causes them to be vulnerable to a similar attack as the one with PKCS#1 v1.5 padding, as shown by Manger[5]. That attack requires as little as $\log_2 n$ oracle calls, where n is the RSA modulus, to perform a ciphertext decryption.

More general plaintext oracle Meyer et al.[6] extended the attack algorithm to knowledge about arbitrary bytes of the message. In their example the oracle responds positively for any PKCS#1 plaintext that starts with arbitrary byte, not just 0x00, but still requires the second byte to be equal 0x02.

Attack summary The different attacks on RSA ciphertexts show that leaking any kind of information about the plaintext allows the attacker to decrypt ciphertexts or sign messages without access to the private key.

In particular, while alternative padding methods, like OAEP, help, they're not a panacea. Processing of any plaintext values, or variables directly related to the plaintext values, still must be performed in a way that does not leak information about the secret values.

3 Performed attacks

As an attacker, we can easily create RSA ciphertexts that decrypt to specific plaintext: by simply encrypting the value we want the deblinding and depadding code to see. By sending such crafted ciphertexts to an implementation under test and measuring the times it takes to process them, we can tell if certain classes of plaintexts don't reveal different code paths taken.

By performing the measurements in double-blind fashion, where neither the test harness not the tested server can guess the PKCS#1 conformance of the decrypted plaintext, we can detect even very small differences in processing time.

3.1 M2Crypto

The M2Crypto library is a thin wrapper around the OpenSSL library. It allows easy access to some of the interfaces of the OpenSSL library from python applications.

One of the APIs supported is the `rsa_private_decrypt`, providing decryption of PKCS#1 v1.5 formatted ciphertexts. Unfortunately, when the underlying OpenSSL API returns an error, M2Crypto translates it to a Python exception (`M2Crypto.RSA.RSAError`). That means that PKCS#1 conforming and PKCS#1 non-conforming ciphertexts will have significantly different code paths executed.

In practical attack, with 1024 bit RSA keys, performed on a regular laptop computer (Lenovo T480s, Intel i7-8650U), with no special configuration and with regular desktop environment running, we were able to differentiate with extremely high confidence (sign test p-values smaller than 10^{60}) conforming and non-conforming ciphertexts by measuring as little as 1000 decryptions of each. This is caused by fairly large median difference between conforming and non-conforming plaintexts, measuring around 0.6µs, when the whole API call takes around 155µs.

With this leak we were able to decrypt a ciphertext using an unoptimised algorithm (i.e. the original one published by Bleichenbacher) in 163 thousand oracle calls, or in about 9h of real time on a regular machine.

The issue was reported to the M2Crypto maintainers in October of 2020 and was assigned the CVE-2020-25657. A partial fix to it was implemented[1] but it does not make the code paths of conforming and non-conforming ciphertexts identical. While we haven't tested this new code, we believe it to still be vulnerable.

[1] https://gitlab.com/m2crypto/m2crypto/-/commit/84c53958def0f510e92119fca14d74f94215827a

3.2 pyca/cryptography

The `pyca/cryptography` is a newer wrapper library providing access to OpenSSL from Python. Similarly to M2Crypto, it too supports the RSA decryption with PKCS#1 v1.5 padding. For that we've used the `decrypt()` method of the `RSAPrivateKey` object.

Just like M2Crypto, `pyca/cryptography` raises an exception in case of malformed PKCS#1 plaintext. That means it is also vulnerable to timing attacks.

We've measured the difference between conforming and non-conforming ciphertexts of around 7.5µs on an Intel 4790K @ 4.4GHz when using 1024 bit RSA keys (with a processing time of about 105µs). With such a huge difference, only 100 measurements were necessary to discern conforming ciphertexts from and non-conforming ones. In practice, on an unoptimised desktop system, with the original Bleichenbacher algorithm (median of 163 thousand oracle calls) the whole decryption took a bit under 4h.

The issue was reported to the `pyca/cryptography` maintainers on 17th of October 2020 as being present in version 3.1.1 of the library. It was assigned the CVE-2020-25659. A partial workaround was developed[2] and shipped as part of version 3.2.

Given that the API throws an exception when OpenSSL returns an error, it's likely still vulnerable; it is now documented though as insecure[3].

3.3 Other high-level language libraries

While we haven't tested other cryptographic library wrappers, we think that any libraries that return errors in fundamentally different way than a successful return from a call will provide a timing oracle for Bleichenbacher like attacks.

Similarly, CVE-2020-25659 and CVE-2020-25657 show that safe use of the RSA PKCS#1 v1.5 API is complex and error-prone.

3.4 NSS

Mozilla NSS is the cryptographic library used by the Firefox browser. As a general-purpose library, it provides support for both TLS ciphersuites that use RSA key exchange and a general purpose API for performing PKCS#1 v1.5 decryption.

One interesting aspect of this library is that it uses a PKCS#11 interface between the implementations of the cryptographic algorithms and rest of the library, like the TLS implementation. PKCS#11 is more commonly used as the API to communicate with cryptographic tokens (like smart cards and hardware security modules).

[2] https://github.com/pyca/cryptography/commit/58494b41d6ecb0f56b7c5f05d5f5e3ca0320d494

[3] `https://cryptography.io/en/latest/limitations/#rsa-pkcs1-v1-5-constant-time-decryption`

We've tested NSS on the TLS level, as that did not require creation of any test harness, effectively performing a black-box test. Additionally, that allowed us to perform an end-to-end test of the whole processing of the secret data: from reading the ciphertext, through decryption, depadding, derivation of the symmetric encryption keys to sending the TLS Alert message.

We've executed the test on a highly optimised machine, setup of which is described in appendix A, with an Intel i9-12900KS.

By running the test on such a system with a 2048 bit RSA key we found (see fig. 2) that the NSS library has very significant leakage, providing 3 easily distinguishable classes of ciphertexts: ones that decrypt to PKCS#1 conforming plaintext with the message being correct size for TLS pre-master secret (48 bytes), ones that decrypt to PKCS#1 conforming plaintext but with incorrect message size (either shorter or longer than 48 bytes), and ones that decrypt to non-conforming plaintexts.

The statistical tests are providing statistically significant results (p-value for sign test smaller than 10^{-4}) for samples that have just 100 observations per class, with Friedman test p-values for the whole test with 31 classes being regularly smaller than 10^{-9} for the same data set.

Based on similar results against version 3.53 of NSS we've informed Mozilla on 16th of June 2020 that the bug #577498[4] is exploitable.

After discussing the possible causes, we've identified the PKCS#11 interface as the culprit. The fact of copying data vs returning an error was causing the significant differences in timing.

We've proposed implementation of an implicit rejection mechanism in the PKCS#11 token, so that it would return a pseudo-randomly generated message based on the received ciphertext and the used private key (the algorithm is described in detail in section 4.2) instead of an error in case of PKCS#1 non-conforming plaintext. That algorithm was implemented in NSS[5] and shipped as part of version 3.61.

While this significantly reduced the observable side-channel (from around 5µs to 60ns), it didn't eliminate it. We've informed NSS developers of this fact on 19th of January 2021.

After some discussions with upstream, we've come to conclusion that the remaining leak is most likely caused the numerical library performing "normalization": making sure that the most significant words of the internal multi-precision integer representation are non-zero after every fundamental operation (like addition or multiplication).

Note that with the Marvin workaround implemented, the testing script needs to have access to the private key to generate ciphertexts that can expose side-channels in the workaround or in the numerical library used for decryption. We describe the Marvin workaround in detail in section 4.2. This is only a verification optimisation, as the test script is reusing the same RSA ciphertexts over and over, similar attack can be performed by randomising the ciphertexts

[4] https://bugzilla.mozilla.org/show_bug.cgi?id=577498

[5] https://phabricator.services.mozilla.com/rNSSfc05574c739947d615ab0b2b2b564f01c922eccd

while keeping specific property of plaintext (like zero most significant bytes) constant. That would require generating unique (or semi-unique) ciphertexts for every connection, which would be slow from the Python test runner we use.

We've executed the test against version 3.80 of the library and identified that plaintexts that have 8 or more zero bytes at the most significant positions have statistically significantly different behaviour (see fig. 3). That confirmed the previous hypothesis about the leak coming from the numerical library. We've informed Mozilla of this on 20th of July 2022. We've also provided to Mozilla on 5th of October 2022 a simple pure C implementation of constant time multiplication and modulo operations (tested on x86_64, ppc64le, aarch64 and s390x architectures) to use for the deblinding operation. That being said, with this smaller side-channel, collecting even 400 thousand observations per class over 22 classes wasn't enough to consistently get statistically significant p-values (smaller than 10^{-5}) from the Freidman test.

As of the time of writing of this article, we're not aware of this, or any other code aiming at performing de-blinding in constant time, being added to NSS.

3.5 OpenSSL

For testing OpenSSL we've been using the same environment as for testing NSS.

Very quickly we've noticed that as few as 10 thousand observations per probe type are enough to have a statistically significant difference (p-values smaller than 10^{-5}) between a probe with many zero values at the most significant bytes and one with PKCS#1 conforming plaintext.

Issue in how BIGNUM is implemented was identified as the primary cause of the vulnerability. Which means that both OpenSSL and NSS suffer from fundamentally the same issue: using a general purpose numerical library to operate on cryptographically sensitive numbers.

Despite the numerical library in OpenSSL having smaller side channel than the one in NSS: OpenSSL is just under 30ns while NSS is about 60ns on the same i9-12900KS CPU; because the OpenSSL responses are quicker (median response of 381µs vs 862µs) and more consistent (MAD of inter-sample differences of 0.357µs vs 13.7µs), the side-channel leakage is easier to detect.

The fix for it (the history of which is described in appendix B) was merged and released as part of version 3.0.8 and 1.1.1t of the library on 7th of February 2023. It was assigned the ID of CVE-2022-4304. We've also verified that the merged patches don't show a side-channel leakage bigger than 10ns when tested on x86_64, ppc64le, s390x and aarch64 architectures.

To fix CVE 2020-25659 in `pyca/cryptography` and CVE 2020-25657 in M2Crypto, we've also proposed to OpenSSL the implementation of the same Marvin workaround as the one implemented in NSS on 8th of January 2021 [6].

[6] `https://github.com/openssl/openssl/pull/13817`

That code was merged to the master branch (intended to become a future 3.2.0 release) on 12th of December 2022[7].

3.6 GnuTLS

While we've also suspected GnuTLS as vulnerable to timing side channels, and informed GnuTLS maintainers about it on 14th of July 2020, this happened before we had a robust approach for measurement though, so identifying a cause from noisy results was difficult.

On 29th of July 2022 Alexander Sosedkin identified a logging function call[8] as likely responsible. We've tested a version of GnuTLS with those lines removed (they're useful only as a debugging aid) and found no side-channel after collecting timings for 34 million connections for each of the 31 types of probes on the highly optimised system with i9-12900KS. Calculated 95% confidence interval for median of differences was ±2ns (so about 10.5 CPU cycles). We've informed GnuTLS developers about this result on 11th of November 2022.

This fix was merged to GnuTLS master[9] and shipped as part of the 3.8.0 release on 10th of February 2023. The issue was assigned a CVE-2023-0361 identifier.

4 Proposed countermeasures

Side-channel signals The implementations of cryptographic algorithms need to both process and generate values in ways that do not leak information about the processed data. There are many different kinds of side-channels: timing, power, sound, light, etc. Generally, when we consider timing attacks, we mean the measurement of the time the whole operation took: how long it took to generate a shared secret, how long a signature operation took, and so on. This kind of side-channels provide only rough information about the processed data or used keys.

For example, a leaky implementation of modular exponentiation, when used together with ciphertext blinding will likely provide information about the Hamming weight of the private exponent or CRT exponents. But Hamming weight alone is insufficient for recovering the private key: Coppersmith method and derived algorithms require knowledge about consecutive bits of at least one private exponent.

Implementations of RSA should thus employ at least ciphertext blinding before performing private key operations. Though, this will only help against the simple timing attack with chosen ciphertexts. For protection against other kinds of side-channels, we recommended additionally use of exponent blinding.

[7] https://github.com/openssl/openssl/commit/7fc67e0a33102aa47bbaa56533eeecb98c0450f7 and following patches

[8] https://gitlab.com/gnutls/gnutls/-/blob/1f0183092125ac3c7449b8ee175f9c303cbab384/lib/auth/rsa.c#L238-245

[9] https://gitlab.com/gnutls/gnutls/-/merge_requests/1698

RSA implementation decomposition When implementing a generic RSA decryption algorithm, used for either RSA key exchange in TLS or called directly by other applications or libraries, multiple things need to happen before the data is securely processed.

1. Modular exponentiation using arbitrary precision integer arithmetic
2. Padding checks and secret extraction (PKCS#1 v1.5 or OAEP)
3. Secret value use and error handling

For RSA specifically, a popular workaround against leaks in the arbitrary precision arithmetic is the use of blinding. With blinding, the ciphertext is multiplied by a random value, the blinding factor. Then such blinded value undergoes modular exponentiation using a regular algorithm. Since the exact value exponentiated is unknown to the attacker, and different for every operation, even with the same ciphertext, they can't infer anything about actual value of it from the timing information alone. But to get access to the actual result of the operation (the encrypted message), the library needs to multiply the result of modular exponentiation by a multiplicative inverse of the blinding factor: the unblinding factor. While the inputs to the unblinding operation are uncorrelated with both the ciphertext and plaintext, and secret to the attacker, the output isn't.

Since CPUs commonly provide instructions to help in multiplication or addition, even if the result doesn't fit into a single register (like 64 bit multiply returning 128 bit result on 64bit CPUs), arbitrary precision implementations commonly store large integers as a list of word-sized integers (where word is the size of biggest general purpose register: 32 bit for 32 bit CPUs, 64 bit for 64 bit CPUs). For a generic purpose numerical library, storing additional words that specify zeros above the most significant digit is useless: it requires more memory and makes computation slower. So generic purpose libraries "clamp" or "normalize" the stored numbers: store only the significant non-zero words.

If that stored number is the result of RSA decryption operation, then difference in number of words used to store it will cause differences in time to convert it into a byte string (which is necessary to test padding, be it PKCS#1 v1.5 or RSA-OEAP, or to feed it into a KDF, like in case of RSASVE). So, by learning that the operation produced a smaller integer, the attacker knows that the high bits were all zero: exactly the information necessary to perform Bleichenbacher or Manger attacks.

4.1 Timing attacks against blinded implementations

Since the leak happens in the very last modular multiplication, the solution for implementations that employ blinding is to implement just that very last operation using constant time code.

While the inputs to that last multiplication come from a general purpose arbitrary precision arithmetic library and thus are clamped; since they are blinded, random, and secret, the conversion of them into constant size representations doesn't have to be side-channel free. Without knowledge of the used blinding factor, knowledge that a particular modular exponentiation provided a small output doesn't provide any information to the attacker.

Once the constant time modular multiplication result is calculated, it needs to be returned as a constant-sized (for a given modulus) list of integers. Converting that list into a byte string of constant size in side-channel free manner is simple.

We were able to implement both the arbitrary precision multiplication and Montgomery reduction algorithms for 64 bit CPUs in just 200 lines of portable C code. We've analysed the generated assembly by both LLVM and GCC compilers and didn't find any data-dependant instructions across code generate by multiple versions of the compilers.

We've also compiled it with GCC 11.3.1-2.1.el9, just with -O3 optimisation level, on x86_64, aarch64, ppc64le, and s390x architectures. We tested its timing characteristics against inputs with very high Hamming weights, very low Hamming weights, with multiple zero words at the beginning and end. We found the code to be completely constant time to the resolution of the best clock source available on each of those platforms.

As such, we believe that implementing side-channel free arbitrary precision integer arithmetic in pure C is possible. Given the speed of the algorithms used for typical cryptographic inputs, we also think that a simple regression test case, to protect against possible compiler optimisations introducing side-channels, executable in a CI environment, is also possible (the tests require less than half an hour of data collection to provide resolution down to single CPU cycles).

4.2 Timing attacks against PKCS#1 v1.5 padding

Application interfaces that implement the PKCS#1 v1.5 padding check are particularly vulnerable. This is caused by three things: the side-channel free check of the padding being complex, extraction and returning of the secret value in side-channel free way to the application being complex, and that handling of the returned error codes and secret value *in the application* needs to be performed in side-channel free manner.

Protocols like TLS work around this problem by performing implicit rejection: when the padding check fails, the size of the extracted secret is wrong, or the protocol version number in the extracted secret is wrong, instead of using extracted value as the secret, they need to use the previously generated random value as the input to the master secret generator function. Since the master secret is calculated from both the extracted pre-master secret and server-selected (outside the attacker's control) random value from the ServerHello message, the attacker is unable to differentiate the decryption failure caused by badly guessed, but actually extracted from PKCS#1 v1.5 ciphertext value, and a previously generated random value[8].

This kind of workaround doesn't work for a generic API, as a randomly generated value will cause a different behaviour of the calling application than a constant value, even if unknown to the attacker. But, since *by definition* the attacker doesn't know if the decrypted value has a PKCS#1 v1.5 compliant padding or not, we can make this signal useless as a Bleichenbacher oracle by making all ciphertexts decrypt to a value, as long as the same ciphertext will decrypt to the same plaintext every time.

One of the features of the PKCS#1 v1.5 padding is that it includes at least 8 bytes of random data as padding. That means that there are almost 2^{64} ciphertexts[10] that decode to one and same message, significantly more if the returned message is smaller. Thus an attacker that has access to the literal result of the decryption would need to encrypt this many ciphertexts to know if the decrypted value could be represented as the given ciphertext, and thus know if the real, padded plaintext starts with a zero byte.

This approach is particularly useful for implementations that expose only PKCS#11 interface, like smart-cards or hardware security modules, as those need to copy different amount of data to the calling application depending on whether the padding check was successful or not.

To calculate an unpredictable, but deterministic, message, we can use the private exponent and the literal ciphertext as the inputs to a key derivation function (similar to the deterministic nonce generation for (EC)DSA signatures[9]).

Note that, as two different implementations of this general idea that use the same key can be used to cross-check if the decrypted value is the result of valid or invalid padding, we strongly recommend to implement the following algorithm precisely as stated. If not done as such, attacks may still be possible against heterogeneous environments.

As such, we propose this alternative algorithm for PKCS#1 v1.5 depadding (the Marvin attack workaround, or implicit rejection for RSA decryption):

1. Check the length of input message according to step one of RFC 8017 Section 7.2.2 (since all inputs are public, this check doesn't have to be performed in side-channel free way and the processing can stop here).
2. Derive the Key Derivation Key (KDK) from the private exponent and public ciphertext
   (a) Convert the private exponent (d) to a big-endian integer, left-padded with zeros so that it has the same size the the public modulus
   (b) Hash it using SHA-256, store that value (since it's constant you can reuse it, but it needs to be kept secret, just like the private exponent)
   (c) Use the hash of the exponent as an SHA-256 HMAC key and the provided ciphertext as a message to the HMAC. The output of the HMAC is the KDK.
3. Create a list of candidate lengths and a random message
   (a) Define a Pseudo Random Function that takes as input a key, label, and number of bytes to output. This function needs to generate sequential blocks of random data by calling SHA-256 HMAC with the provided key as the key, and message set to concatenation of an iterator (initialised to 0, increased by 1 for every HMAC call, encoded as a two-byte big-endian integer), the label (as-is, *without* C-like null byte termination) and the number of bits to output (i.e. 8 times the number of output

[10] exactly it is equal to $(2^8 - 1)^8 \approx 2^{63.95}$, as every individual byte of the padding must not be equal 0 and there are 8 of them

   bytes; encoded as two-byte big-endian integer). If output size is not a multiple of SHA-256 HMAC size, the output should be right-truncated to fit (i.e. only the most significant bytes of last HMAC output should be returned).
   (b) Using the PRF with KDK and "length" as six byte label encoded with UTF-8 generate 256 byte output. Interpret it as 128 two byte big-endian numbers.
   (c) Using the PRF with KDK and "message" as seven byte label encoded with UTF-8 generate as many bytes as are necessary to represent the modulus (k). This is the alternative decryption to use in case the padding check fails.
4. Select a length of the returned message in case the padding check fails (Note: this step needs to be performed in side-channel free way)
   (a) For each of the 128 possible lengths zero-out the high-order bits so that they have the same bit length as the length of the maximum acceptable message size ($k - 11$).
   (b) Select the last length that's not larger than $k - 11$, use 0 if none are.
5. Perform standard RSA decryption as described in step 2 of RFC 8017 section 7.2.2. (Note: this step needs to use side-channel free code)
6. Verify the EM padding as described in step 3 of RFC 8017 section 7.2.2, but instead of outputting "decryption error", return the last l bytes of the "message" PRF, where l is the selected length from step 4. (Note: both selection of use of the generated message as well as the copy of it needs to be performed with side-channel free code).

Practical implementations as well as test vectors of this algorithm can be found in tlslite-ng (pure Python), Mozilla NSS, and OpenSSL PR #13817.

While this algorithm changes the semantics of error handling, so code that depends on "decryption error" to mean that the key used to decrypt the ciphertext was incorrect may misbehave, it should be noted that a plaintext returned by decrypting a ciphertext under a different key-pair that was used to encrypt it will be effectively random. Random plaintexts have a non irrelevant chance of being PKCS#1 v1.5 conforming. Thus the use of this alternative algorithm changes the *likelihood* of getting a message decryption by using a wrong key, it doesn't change the *possibility* of it. So any protocol that tries decryption of RSA ciphertexts with different keys needs to employ a different way to detect if the ciphertext matches the key than the absence of errors in RSA PKCS#1 v1.5 decryption.

4.3 Countermeasures summary

While we provide an algorithm for more secure PKCS#1 v1.5 depadding, given the complexity of implementing and testing this algorithm, we strongly recommend for libraries to instead remove support for encryption using PKCS#1 v1.5 padding completely. The far simpler workaround described for TLS (section 7.4.7.1 of RFC 5246[8]) was previously found to be implemented incorrectly by over 20 different implementations[3]. That's on top of the fact that testing for correctness of the TLS-specific workaround is much easier than testing for correctness of the Marvin workaround. Thus, we would consider any use of generic PKCS#1 v1.5 API that doesn't use the Marvin workaround internally to be a case of CWE-242[11] ("Use of Inherently Dangerous Function") and, without a verified side-channel free code on the calling side, an automatic vulnerability for the calling code.

While we haven't tested any actual hardware PKCS#11 modules, based on results from NSS, we're afraid that most, if not all uses of PKCS#11 tokens and modules for PKCS#1 v1.5 decryption will be vulnerable to the Bleichenbacher oracle in practice. Simply transferring a different amount of data flowing between application and module in case of an error and a message that decrypts to very few or very many bytes would already provide enough of a side-channel to make it vulnerable.

We also recommend against the use of OAEP and RSASVE with libraries that don't have verified side-channel free arbitrary precision integer arithmetic library. Any library that uses general purpose integer arithmetic implementations should be considered suspect.

5 Test framework

To conduct those tests we've used the `tlsfuzzer` test suite [12]. It's a TLS protocol conformity test suite able to generate different kind of arbitrary messages to send to the server and then to verify that the reply matches some expectations.

It can perform normal handshakes, exchange data and perform orderly connection close of any protocol version between SSL 2 and TLS 1.3, or inject errors at any point in the connection to test server's error handling.

We've used it to send the pregenerated RSA ciphertexts to the TLS server in the TLS ClientKeyExchange messages.

For performing the general tests `tlsfuzzer` has minimal dependencies: only the Python environment itself and few pure-python libraries (`tlslite-ng`, `python-ecdsa`, and `six`). It can run on Python 2.6, 2.7, 3.5 or later. For timing data collection it additionally requires `dpkt` and permissions to run `tcpdump`. As such, collecting data should be possible on any actively supported operating system. Later analysis of the data requires packages like `numpy`, `scipy`, and `pandas`, limitin the supported Python version to 3.6 as the oldest.

For conducting the timing tests, the scripts can first generate the test payloads in random order, write them to disk, together with information which payload corresponds to which probe. Such generated payloads are then read sequentially, send one by one to the server, and server response times are captured by the `tcpdump` running in the background.

This ensures that the payload generation, its name, placement in memory, or anything similar, doesn't influence the timing of probe sending, making the

[11] `https://cwe.mitre.org/data/definitions/242.html`

[12] `https://github.com/tlsfuzzer/tlsfuzzer`

test harness effectively constant time. By using packet capture to collect timing data, we both provide larger separation between measuring server response times and ensure that the payloads sent by the test harness don't influence the measurement.

Only when the individual time measurements are interpreted according to the order in which they were originally generated do the patterns in server responses emerge.

The script we generally used to perform those tests is the `test-bleichenbacher-timing-pregenerate.py` in the `scripts` directory of the tlsfuzzer repo. To collect timing data, at the very least the output directory (with `-o`) and the network interface on which to perform capture (using `-i`) must be provided. See its `--help` message on more tips on its execution.

For servers that implement the Marvin workaround on the API level, we have prepared a script that generates ciphertexts decrypting to the same length of plaintext both for valid and invalid padding case: the `test-bleichenbacher-timing-marvin.py`.

The framework includes also two scripts for analysing multiple individual test script executions. The `combine.py` in the `tlsfuzzer` directory can be used to combine data from multiple runs (provided that the same set of probes was used in all the runs). Such combined data set can then be analysed using the `analysis.py` script in the same directory. See their help messages about supported options. When analysing large data sets (above 100k observations per sample) we recommend disabling generation of additional graphs though command line options.

More information about executing timing tests is available in the tlsfuzzer documentation[13].

Based on tlsfuzzer code we've also created a set of scripts for preparing test cases for testing generic RSA encryption functions as the `marvin-toolkit` [14].

It should be noted that it does not create random, single use ciphertexts, like the TLS script, so trying to measure decryption of the same ciphertexts over and over may report false positives if the numerical library is not fully constant time (as then the leak based on *ciphertext* may end up being detected, which is not security relevant).

6 Future work

In this work, we have focused only on the simplest side-channel attack: a low granularity timing side channel. Higher granularity side-channels, like ones from microarchitectural sources, together with more robust statistical methods, are likely to show that fewer observations are necessary for statistically significant results. More advanced side-channel attacks, like ones that use power analysis, electromagnetic emissions, or sound are still likely possible.

[13] `https://tlsfuzzer.readthedocs.io/en/latest/timing-analysis.html`

[14] `https://github.com/tomato42/marvin-toolkit`

We have tested just a handful of the popular cryptographic libraries. Larger scale testing of software and hardware implementing RSA encryption (of any kind) will likely reveal many more vulnerable implementations.

Only the Bleichenbacher attack against RSA decryption was tested. Performing similar tests against constant-timeness with regards to used private keys should also be possible with similar approach and a proper test harness.

Extending the presented approach should also be possible for testing other timing attacks in TLS, like the Lucky13 attack.

7 Summary and recommendations

We've shown that by using correct statistical methods we can detect much smaller timing side-channels than previously expected to be possible.

With this new approach we've analysed multiple cryptographic libraries, both ones implementing the algorithms directly (OpenSSL, NSS, and GnuTLS), as well as higher-level language bindings (M2crypto, and pyca/cryptography). Every single one of them turned out to be vulnerable or exploitable to the Bleichenbacher attack against RSA encryption. Our recommendation is thus that RSA encryption shouldn't be used, as implementing it correctly is very hard, if not impossible. We especially recommend that the PKCS#1 v1.5 padding for RSA encryption should not be used, and any protocols that allow its use should deprecate, forbid its use completely.

For implementations that cannot deprecate and remove support for PKCS#1 v1.5 decryption we've proposed an algorithm to implement implicit rejection of ciphertexts that fail the padding check. We recommend its use in all general APIs that cannot remove support for PKCS#1 v1.5 decryption, including PKCS#11. We must stress though, that implementing it correctly and verifying correctness of that implementation is hard, so it should be employed as a last-ditch solution, when all other options to remove need for PKCS#1 v1.5 encryption have been exhausted.

We recommend that static code analysis scanners should mark any uses of PKCS#1 v1.5 decryption APIs as inherently unsafe.

We've also shown that while the use of mitigations such as (base) blinding for RSA decryption helps, it cannot be implemented blindly and steps that have access to real plaintext values, like the unblinding step and conversion from multi-precision integer to a byte string, must be implemented with special care and with verified side-channel free code. We recommend to consider any implementation of cryptographic arithmetic that uses general-purpose multi-precision numerical methods to be vulnerable to side-channel attacks. In particular, any code that uses variable size internal representation of integers is, most likely, vulnerable to side-channel attacks.

8 Acknowledgments

I'd like to thank Jan Koscielniak for the initial test implementation and test results that were the inspiration for this research. Stefan Berger for discussions that led to the workaround on API level. Daniel J. Bernstein and Juraj Somorovsky for research pointers and sanity check of the workaround idea. Greg Sutcliffe for discussions about statistical methods for analysing the timing data.

9 References

[1] Romain Bardou et al. "Efficient Padding Oracle Attacks on Cryptographic Hardware". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 608–625. ISBN: 978-3-642-32009-5.

[2] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1." In: *CRYPTO*. Ed. by Hugo Krawczyk. Vol. 1462. Lecture Notes in Computer Science. Springer, 1998, pp. 1–12. ISBN: 3-540-64892-5. URL: `http://dblp.uni-trier.de/db/conf/crypto/crypto98.html#Bleichenbacher98`.

[3] Hanno Böck, Juraj Somorovsky, and Craig Young. "Return Of Bleichenbacher's Oracle Threat (ROBOT)". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 817–849. ISBN: 978-1-939133-04-5. URL: `https://www.usenix.org/conference/usenixsecurity18/presentation/bock`.

[4] Vlastimil Klíma, Ondrej Pokorný, and Tomáš Rosa. "Attacking RSA-Based Sessions in SSL/TLS". In: *Cryptographic Hardware and Embedded Systems - CHES 2003*. Ed. by Colin D. Walter, Çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 426–440. ISBN: 978-3-540-45238-6.

[5] James Manger. "A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 230–238. ISBN: 978-3-540-44647-7.

[6] Christopher Meyer et al. "Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 733–748. ISBN: 978-1-931971-15-7. URL: `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer`.

[7] B. Kaliski and J. Staddon. *PKCS #1: RSA Cryptography Specifications Version 2.0*. RFC 2437 (Informational). RFC. Obsoleted by RFC 3447. Fremont, CA, USA: RFC Editor, Oct. 1998. DOI: `10.17487/RFC2437`. URL: `https://www.rfc-editor.org/rfc/rfc2437.txt`.

[8] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). RFC. Obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447, 9155. Fremont, CA, USA: RFC Editor, Aug. 2008. DOI: `10.17487/RFC5246`. URL: `https://www.rfc-editor.org/rfc/rfc5246.txt`.

[9] T. Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979 (Informational). RFC. Fremont, CA, USA: RFC Editor, Aug. 2013. DOI: `10.17487/RFC6979`. URL: `https://www.rfc-editor.org/rfc/rfc6979.txt`.

[10] M. Jones and J. Hildebrand. *JSON Web Encryption (JWE)*. RFC 7516 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, May 2015. DOI: `10.17487/RFC7516`. URL: `https://www.rfc-editor.org/rfc/rfc7516.txt`.

[11] K. Moriarty (Ed.) et al. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017 (Informational). RFC. Fremont, CA, USA: RFC Editor, Nov. 2016. DOI: `10.17487/RFC8017`. URL: `https://www.rfc-editor.org/rfc/rfc8017.txt`.

[12] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Aug. 2018. DOI: `10.17487/RFC8446`. URL: `https://www.rfc-editor.org/rfc/rfc8446.txt`.

A System tuning

To minimise amount and magnitude of the noise in measurements we found some changes to system configuration to be very effective.

The BIOS was configured to override the processor base power to the same level as the maximum turbo power (241W), so as to remove the time limits on how long will the CPU run with turbo boost (run at elevated frequency). The BIOS was also configured to allow high frequency (high multiplier) operation even when multiple cores are active (we've noticed that this is important as the BIOS/CPU consider the core to be "active" when it's in the C2 power state or higher).

Hyper-Threading was disabled. The Linux kernel was configured using the `tuned` cpu-isolation profile with 4 of the 8 P-cores isolated. Tuned cpu-isolation profile sets the idle driver to keep all the CPU cores (not just the isolated ones) at the C1 power state. This is important because the test harness (`tlsfuzzer`) and the system under test (like NSS `selfserv` or openssl `s_server`) execute on separate cores and use a network protocol to communicate, so there are idle periods when they wait for a reply from the other side of the connection. During those idle periods, the CPU normally goes into a deeper idle state (lower power state): C2, C3, or higher. The problem is that going out of those idle states back to the state where the CPU can execute instructions (C0) takes different amounts of time, generally the deeper the C-state, the longer the transition to C0 state. C1 state is a bit special in that it's reported by the hardware as requiring just a single CPU cycle to transition to C0. In quick testing we haven't noticed qualitatively better results by disabling C-states completely and using just the Linux polling idle driver compared to the approach taken by `tuned`. At the same time, allowing the CPU to switch to C3 states did cause the results to be significantly worse, increasing the bootstrapped 95% confidence interval of the median of differences from 0.223µs to 3.23µs and the median absolute deviation[15] of inter-sample differences from 7µs to 1.2ms.

The machine also has configured aggressive fan curves and a large CPU heatsink installed, causing the CPU to stay under 50°C when running the tests, often around 40°C, making sure that the CPU does not employ thermal throttling.

The CPU was running at a stable 5.225GHz when measuring the server response times. We also tested a configuration in which the two cores used for measurement were running at the maximum supported frequency of 5.5GHz, but found it to provide lower quality results, not offset by the quicker execution.

Please note that while this configuration provides higher quality results, it's *not* necessary for the correct operation of the statistical tests.

B OpenSSL fix history

The development and integrations of the patches to the OpenSSL took a very long time.

We've originally informed the OpenSSL project that their implementation of RSA decryption in version 1.1.1c is vulnerable on 14th of July 2020.

Over the next few weeks (on 6th of August) we've identified the previously reported issue #6640[16] (in the way that BIGNUM code is implemented) as the primary cause of the timing side channel.

On 15th of July 2022 we've informed OpenSSL that the implementation is most likely exploitable against a network attacker when non standard key sizes (2049 bit or 2056 bit) or 32 bit compiles are used. In that message we've also suggested workarounding the leakage in BIGNUM implementation by performing the deblinding step using a portable C implementation of multiplication and modulo operations. See section 4.1 for details.

The code to perform that, including one that uses Montgomery reduction to calculate the mod was provided to OpenSSL in October 2022.

C Graphs of test results

[15] Median absolute deviation (MAD) is a robust measure of the variability of the data, similar to standard deviation measure, but resilient against outliers

[16] `https://github.com/openssl/openssl/issues/6640`

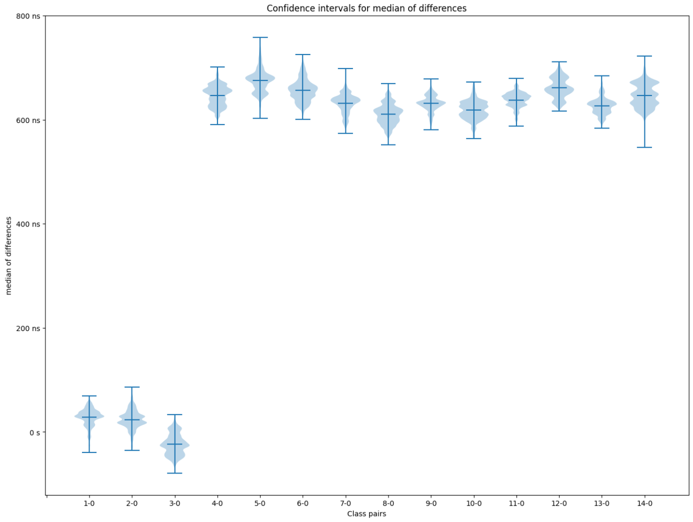

Fig. 1: Bootstrapped confidence intervals of median of differences of different PKCS#1 conforming (probes 1, 2, and 3) and non-conforming plaintexts (4 and larger) compared to a PKCS#1 conforming plaintext. M2Crypto 0.35.2, Intel i7-8650U, 1000 observations per class.

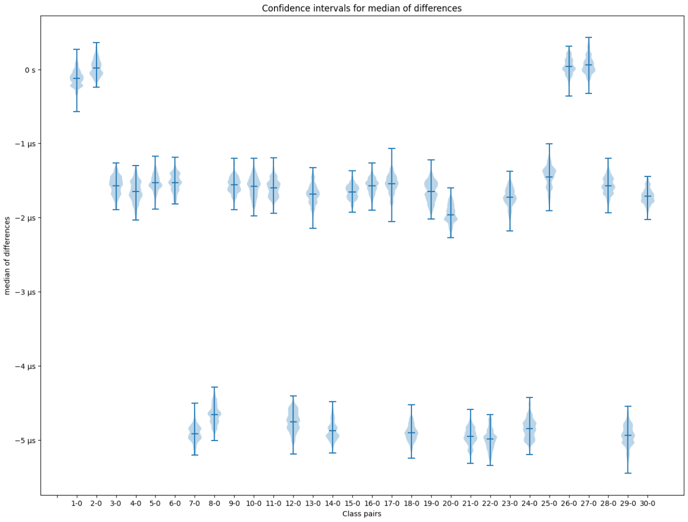

Fig. 2: Bootstrapped confidence intervals of median of differences of different PKCS#1 conforming (probes 1 and 2), conforming but with wrong TLS version (probes 26 and 27), conforming but with wrong encrypted message length for the TLS pre-master secret (probes 7, 8, 12, 14, 18, 21, 22, 24, and 29) and non-conforming plaintexts (remaining) compared to a PKCS#1 conforming plaintext. NSS 3.60, Intel i9-12900KS, 10000 observations per class.

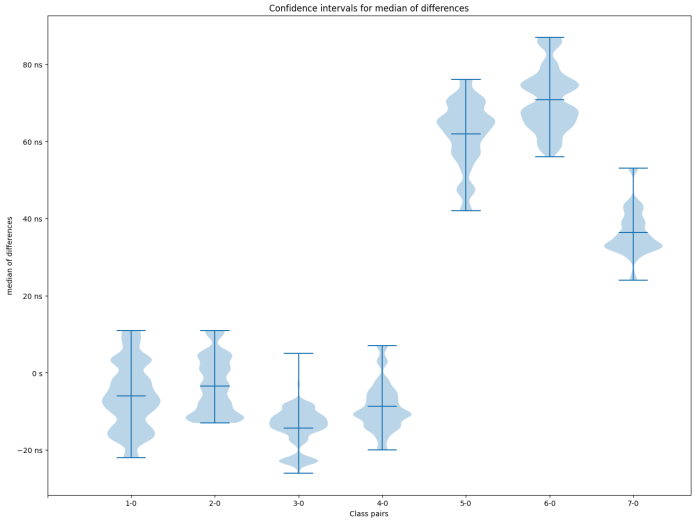

Fig. 3: Bootstrapped confidence intervals of median of differences of different PKCS#1 non-conforming probes compared to a PKCS#1 non-conforming plaintext. The probe 2 has all bytes non zero, probe 1 has one most significant byte set to zero, probe 3 has two, 4 has four, probe 5 has 8 zero bytes, probe 6 has 16, and 7 has 40 most significant bytes set to zero. NSS 3.80, Intel i9-12900KS, 33.5 million observations per class.

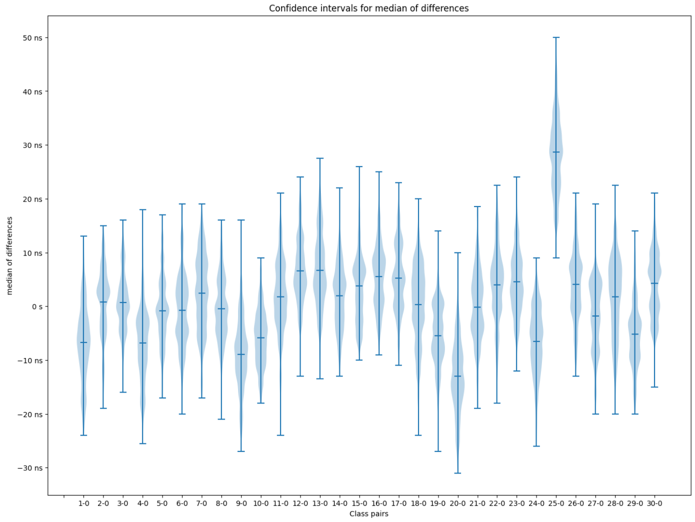

Fig. 4: Bootstrapped confidence intervals of median of differences of different probes compared to a PKCS#1 conforming plaintext. The probe 25 has forty of the most significant bytes set to zero. OpenSSL 1.1.1p, Intel i9-12900KS, 10 thousand observations per class.