# Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks

*Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk*
*Horst Görtz Institute for IT-Security, Ruhr-University Bochum*
*<{christopher.meyer, juraj.somorovsky, eugen.weiss, joerg.schwenk}@rub.de>*

*Sebastian Schinzel <schinzel@fh-muenster.de>*
*Department of Computer Science, Münster University of Applied Sciences*

*Erik Tews <erik@datenzone.de>*
*European Center for Security and Privacy by Design,Technische Universität Darmstadt*

## Abstract

As a countermeasure against the famous Bleichenbacher attack on RSA based ciphersuites, all TLS RFCs starting from RFC 2246 (TLS 1.0) propose "to treat incorrectly formatted messages in a manner indistinguishable from correctly formatted RSA blocks".

In this paper we show that this objective has not been achieved yet (cf. Table 1): We present four new Bleichenbacher side channels, and three successful Bleichenbacher attacks against the *Java Secure Socket Extension (JSSE)* SSL/TLS implementation and against hardware security appliances using the *Cavium NITROX SSL accelerator chip*. Three of these side channels are timing-based, and two of them provide the first timing-based Bleichenbacher attacks on SSL/TLS described in the literature. Our measurements confirmed that all these side channels are observable over a switched network, with timing differences between 1 and 23 microseconds. We were able to successfully recover the `PreMasterSecret` using three of the four side channels in a realistic measurement setup.

| TLS impl. | Side channel | Queries & Efficiency | |
|---|---|---|---|
| | | Queries | Time |
| OpenSSL | timing | $O(2^{40})$ | n.a. |
| JSSE | error message | 177,000 | 12 h |
| JSSE | timing | 18,600 | 19.5 h |
| Cavium | timing | 7371 | 41 h |

Table 1: Overview on Bleichenbacher side channels and attacks. In case of timing based side channels, *Queries* denotes the number of queries sent to the Bleichenbacher oracle $\mathcal{O}$ (see below); the actual number of requests sent to the TLS server (and thus the attack duration) depend on the network quality. Even though we found timing differences in the OpenSSL implementation, the attack revealed not to be practical due to the weakness of the oracle.

## 1 Introduction

SSL/TLS is, due to its enormous importance, a major target for attacks. During the last years, novel attack techniques (targeting the TLS Record Layer) have been discovered (see e.g. [21]). However, one of the most famous attacks is still Bleichenbacher's chosen-ciphertext attack on the TLS handshake [5], exploiting side channels of the RSA decryption process (see Section 3). Formal models don't cover this attack: The first full security proof of the TLS-RSA handshake [17] assumes that the RSA decryption implementation is ideal without any side channels.

**Bleichenbacher's Attack.** Bleichenbacher's attack is an adaptive chosen-ciphertext attack on the RSA PKCS#1 v1.5 encryption padding scheme (denoted by PKCS#1 in the following). The only prerequisite for the attack is the presence of a side channel at the TLS server which allows to distinguish PKCS#1 compliant from non-compliant ciphertexts. An attacker with access to such a side channel can proceed as follows: He records the TLS handshake of the target connection, and extracts the RSA-PKCS#1 encrypted `ClientKeyExchange` message $c$. Then he iteratively creates new ciphertexts $c', c'', \ldots$ from $c$. These are sent to the TLS server as part of a new handshake, and the server's responses are observed. With each successful query, i.e. a query $c^*$ which is PKCS#1 compliant, the attacker can reduce the interval in which the original plaintext is located in. He repeats these steps until the interval only contains one integer, thus decrypting the ciphertext $c$. Daniel Bleichenbacher successfully applied this attack to SSL 3.0 [5] in 1998.

In three of the four presented attacks we are dealing with timing based side channels, so we have to repeat measurements to statistically eliminate random noise. In the following, we use an abstraction to deal with this fact: A Bleichenbacher *oracle* $\mathcal{O}$ receives a candidate ciphertext $c^*$ as input and makes use of a side channel (e.g. by

repeating measurements) to finally output whether $c^*$ is PKCS#1 compliant or not (see Figure 3).

**Countermeasures.** Soon after the publication of the original Bleichenbacher attack in 1998, error messages were unified and the TLS standards introduced the following countermeasure: If the decrypted message structure is not compliant, the TLS server generates a random `PreMasterSecret`, and performs all subsequent handshake computations with this value.[1] This countermeasure was described in TLS versions 1.0 [9] and 1.1 [10]. TLS 1.2 [11] improves this by prescribing that a random number must *always* be generated, independently of the PKCS#1 compliance of the incoming ciphertext. This should ensure equal processing times for compliant and non-compliant ciphertexts.

**Novel Side Channels.** In this paper we analyze several widely used TLS implementations for their vulnerability against Bleichenbacher attacks and show that the implemented countermeasures are not sufficient: We describe four new Bleichenbacher oracles, and analyze their sources (see Table 1). Additionally, the strength of these oracles is evaluated and three of these oracles are shown to be strong enough to mount Bleichenbacher attacks in practice. This finally led to the decryption of previously recorded SSL/TLS sessions.

The first side channel is caused by an implementation bug in the *Java Secure Socket Extension (JSSE)* – Java's built-in SSL/TLS implementation. In JSSE a different error message can be triggered if the two most significant bytes are PKCS#1 compliant, but the `PreMasterSecret` shows up to be of invalid length. We were able to successfully exploit this and decrypt a `PreMasterSecret` with a few thousand queries.

The second side channel is based on conspicuous timing differences in the *OpenSSL* implementation during PKCS#1 processing. The source of this side channel is hard to determine: Our working assumption suggests that it is based on the additional time consumption of choosing a random value. Following the description of Bleichenbacher countermeasures in TLS versions 1.0 and 1.1, this random value is only generated if the decrypted `PreMasterSecret` is not PKCS#1 compliant. The timing difference (in the range of few microseconds) caused by the unequal treatment of random number generation (depending on the PKCS#1 compliance of the ciphertexts) may be the cause for this side channel. We were able to reliably measure a timing difference in the range of one microsecond over a LAN and to reliably detect plaintexts containing valid `PreMasterSecret` values.

The third side channel is based on the fact that Java's `Exception` handling and error processing can be a time consuming task: Whenever the resulting plaintext is not PKCS#1 compliant, an `Exception` is raised by *JSSE* forcing random `PreMasterSecret` generation. The resulting timing difference is significantly higher (in the range of 20 microseconds) and can be measured over a LAN. This qualifies the side channel for practical attacks under real-world conditions.

The fourth side channel was found in widely used *F5 BIG-IP* and *IBM Datapower* products which rely on the *Cavium NITROX SSL accelerator chip*. It allowed to distinguish invalid messages from messages starting with `0x??02` (where `0x??` represents an arbitrary byte). Since the original Bleichenbacher algorithm does not handle this case, we derived a novel variant of the algorithm and evaluated that it can decrypt 2048-bit ciphertexts with only 4700 queries to an oracle.

**Contribution.** The contributions of this paper can be summarized as follows:

- *Impact.* We analyze several widely used SSL/TLS implementations and identify four new Bleichenbacher side channels, three of them timing-based. We describe three successful Bleichenbacher attacks which completely break JSSE and NITROX based SSL/TLS accelerators.

- *Novelty.* We describe the first timing based Bleichenbacher attacks against a TLS implementation. We present a novel variant of the original Bleichenbacher algorithm to handle specific server behavior and show that this variant results in a much better attack performance.

- *Insight.* We show that `Exception` handling may cause large timing differences, measurable over a LAN. This observation is in general important for development of side channel free (cryptographic) implementations in object oriented languages.

- *Methodology.* Our research was conducted using a novel framework for SSL/TLS inspection and penetration, called T.I.M.E., which may be of independent interest.

**Responsible Disclosure.** All vulnerabilities were communicated to the vendors' security teams and sent together with fix proposals. They were fixed or are going to be fixed in the newest releases.

[1] This leads to a fatal error when checking the `ClientFinished` (because of different `PreMasterSecret` at client and server side), but it does not allow the attacker to distinguish valid from invalid ciphertexts based on server error messages.

## 2 SSL/TLS

The Secure Sockets Layer (SSL) protocol was invented 1994 by Netscape Communications, and later (1999) renamed to Transport Layer Security (TLS) by the IETF. It evolved to be the de facto standard for secure data transmission over the Internet and is mostly used, but not limited, to secure HTTP traffic.

SSL/TLS mainly consists of two components: the *Handshake Protocol* to negotiate security primitives and key material, and the *Record Layer* where the payload (HTTP, IMAP, ...) is encrypted and integrity protected.

**Record Layer.** The Record Layer is initiated with the `NULL` ciphersuite, where no cryptographic protection is applied at all. Then the handshake is executed, until the `ChangeCiperSpec` message is sent by one party. Immediately after sending this message, this party switches the Record Layer to the negotiated parameters (algorithms and keys) and enables the negotiated security algorithms.

Subsequently, all messages sent through the TLS channel are secured by the selected cipher suite algorithms and the computed key material. Regarding integrity and confidentiality the Record Layer relies on a MAC-then-PAD-then-Encrypt scheme ([22] gives a detailed overview on this topic and highlights the pitfalls). The payload data is integrity protected by a (keyed H)MAC, padded if required, and finally encrypted.

**Handshake Protocol.** This protocol is used to negotiate the cryptographic primitives and keys. The different primitives are bundled in *cipher suites*. A *cipher suite* defines the algorithms for *(a)* key exchange or key agreement, *(b)* encryption (and, if necessary, the mode of operation) and *(c)* MAC (Message Authentication Code). Thus, the cipher suite `TLS_RSA_WITH_DES_CBC_SHA` uses (a) RSA encryption for key exchange, (b) DES encryption in CBC mode for encryption and (c) a SHA-1 based HMAC for integrity to protect the payload.

Figure 1 illustrates a typical (RSA-based) handshake without mutual authentication, between a client and a server. All cipher suites supported by the client are listed in the `ClientHello` message, and one of these suites is chosen by the server in the `ServerHello` message. The server's public key for RSA encryption is sent in the `Certificate` message and the ciphertext of the `PreMasterSecret` chosen by the client is contained in the `ClientKeyExchange` message. After this message, both - client and server - are ready to switch to encrypted mode (by sending a `ChangeCiperSpec` message). The final two *`Client-/Server`*`Finished` messages (containing a cryptographic checksum over all previously exchanged handshake messages) are already encrypted.

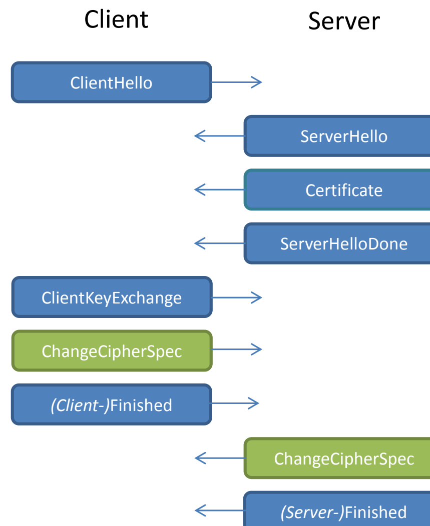


Figure 1: SSL/TLS handshake with RSA Key Exchange

**Other Protocols.** The *ChangeCipherSpec Protocol* is used to activate channel protection (switch to negotiated cipher suite and related key material), whereas the *Alert Protocol* is responsible for signalizing errors and failures.

**Libraries and Appliances.** The work presented in this paper focuses on the most common open source libraries and SSL/TLS appliances listed below.

**OpenSSL.**[2] As a widely used open source library OpenSSL is applied by many applications (such as the Apache Webserver's default module for SSL/TLS).

**Java Secure Socket Extension (JSSE).**[3] This library is the standard implementation of SSL/TLS for the Java platform, provided as part of the Java Runtime Environment. Java based applications are very likely to use it.

**GnuTLS.**[4] GnuTLS is another open source library for SSL/TLS available under GPL.

***IBM Datapower* and *F5 BIG-IP*.** These two products are widely used Web application firewalls and security appliances. Their SSL/TLS processing is handled using a *Cavium NITROX SSL accelerator chip*.

## 3 Bleichenbacher's Attack

In 1998, Daniel Bleichenbacher presented an adaptive chosen-ciphertext attack on protocols using the RSA PKCS #1 encryption standard [5]. He exemplarily applied his attack to the SSL v3.0 protocol. Through different error messages returned from the SSL server, Blei-

[2] `http://www.openssl.org`

[3] `http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html`

[4] `http://www.gnutls.org`

chenbacher was able to identify ciphertexts where the plaintext started with 0x00 02. Thus, he used the SSL server as a (partial) decryption oracle $\mathcal{O}$ and was able to decrypt an encrypted PreMasterSecret, from which all SSL/TLS session keys are derived [11]. Soon after this discovery, the error messages were unified in order to close this side channel. Later, the attack was reenabled by Klíma, Pokorný and Rosa [16] through a different side channel, fixed again and finally remained unexploitable for nearly 10 years.

In order to describe the basic attack, we will first give an overview of the PKCS#1 encryption padding scheme and its usage in SSL/TLS to secure the PreMasterSecret. Afterwards, the attack and the countermeasures are presented. Throughout this section we write $|a|$ to denote the byte-length of a string $a$, and $a||b$ to denote concatenation of $a$ and $b$. We let $(N,e)$ be an RSA public key, with corresponding secret key $d$.

## 3.1 PKCS#1 v1.5 Encryption Padding

**PKCS#1 v1.5.** The basic task of the PKCS#1 v1.5 encryption padding scheme is to prepend a random padding string $PS$ ($|PS| > 8$) to a message $k$, and then apply the RSA encryption function:

1. The encrypter takes a message $k$ and chooses a random, non-zero string $PS$, where $|PS| > 8$ and $|PS| = \ell - 3 - |k|$.
2. The cleartext block is $m = 00||02||PS||00||k$. By interpreting this string as an integer $m < N$,
3. the ciphertext is computed as $c = m^e \bmod N$.

To decrypt such a ciphertext, the decrypter first computes $m = c^d \bmod N$. Afterwards, it is checked whether the decrypted message $m$ has a correct PKCS#1 format. This message $m = m_1||m_2||...||m_{|m|}$ is PKCS#1 compliant if ($x \geq 10$):

$$\begin{aligned} m_1 &= \texttt{0x00} \\ m_2 &= \texttt{0x02} \\ \texttt{0x00} &\notin \{m_3,\ldots,m_x\} \\ \texttt{0x00} &\in \{m_{x+1},\ldots,m_{|m|}\} \end{aligned} \tag{1}$$

**PKCS#1 usage in TLS.** In case of TLS, PKCS#1 is used for encapsulation of the PreMasterSecret exchanged during a handshake which consists of 48 bytes. The first two bytes of the PreMasterSecret contain a two-byte version number $maj||min$ (e.g., $maj = 0x03$, $min = 0x01$ for TLS 1.0). The remaining bytes are chosen by the client at random. Figure 2 gives an example of a PreMasterSecret ($PMS$) padded to be encrypted with a 2048-bit RSA key.

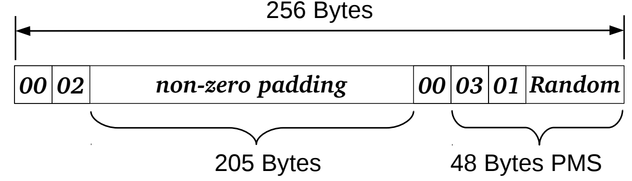


Figure 2: PKCS#1 padding applied to a PMS to be encrypted with a 2048-bit RSA key

We say that a PKCS#1 compliant message $m$ is *TLS compliant* if:

$$\begin{aligned} |k| &= 48 \\ k_1||k_2 &= maj||min \end{aligned} \tag{2}$$

## 3.2 Basic Attack Idea.

Bleichenbacher's attack enables an adversary, who is in possession of a ciphertext $c_0$, to recover the encrypted plaintext $m_0$. The only prerequisite for this attack is the ability to access an oracle $\mathcal{O}$ that decrypts a ciphertext $c$ and responds with 1 or 0, depending on whether the decrypted message $m$ starts with 0x00 02 or not:

$$\mathcal{O}(c) = \begin{cases} 1 & \text{if } m = c^d \bmod N \text{ starts with } \texttt{0x0002} \\ 0 & \text{otherwise.} \end{cases}$$

If the oracle answers with 1, the adversary knows that $2B \leq m \leq 3B-1$, where $B = 2^{8(\ell-2)}$. The algorithm is based on the malleability of the RSA encryption scheme which allows the following blinding:

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

The attacker queries the oracle with $c$. If the oracle responds with 0, the attacker increments $s$ and repeats the previous step. Otherwise, the attacker learns that

$$2B \leq m_0 s - rN < 3B$$

for some $r$. This allows the attacker to reduce the set of possible solutions to

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

By iteratively choosing new values for $s$, querying the oracle, and computing new $r$ values, the attacker narrows down the interval which contains the original $m_0$ value. He repeats these steps until only one solution in the interval is left. We refer to the original paper [5] for details.

### 3.3 Impact of Oracle Type on Attack Performance

The oracle $\mathcal{O}$ needed for the attack can be based on different side channels. For example, it can be provided by a server responding with different error messages based on the PKCS#1 compliance. If the server identifies a message as PKCS#1 compliant, the attacker knows the message starts with 0x00 02.

Bleichenbacher tested his attack against an SSL server which strictly checked the PKCS#1 format (see Equation 1). He needed about one million messages to decrypt an arbitrary ciphertext (1024-bit RSA). However, the attack performance varies. Bleichenbacher's algorithm relies solely on the knowledge that the first two message bytes are equal to 0x00 02. If an oracle is constructed from an application which verifies only the first two bytes of the decrypted message (0x00 02), we get a very "strong" oracle and the attack performs well. On the other hand, if an application checks also different properties such as TLS protocol version conformity (see Equation 2), the oracle can respond with 0 even if the first two bytes are equal to 0x00 02 (e.g., if the extracted `PreMasterSecret` is of invalid length). Such a behavior leads to false negatives which slow down the attack performance. The oracle is "weak".

The oracle strength can be measured using a probability that the oracle responds with 1 when a given decrypted message starts with 0x00 02. Suppose $P(A)$ defines a probability that the first two bytes of the decrypted message are 0x00 02. $P(1|A)$ is a probability that the oracle answers with 1, in case that the decrypted message starts with 0x00 02. Suppose we work with a 1024 bit RSA key. For an oracle strictly checking the PKCS#1 compliance (first eight bytes do not contain 0x00, but one of the following 118 bytes contains 0x00), the probability can be computed as:

$$P_{PKCS}^{1024}(1|A) = \left(\frac{255}{256}\right)^8 \cdot \left(1 - \left(\frac{255}{256}\right)^{118}\right) \approx 0.36$$

Different oracle types and their impact on the attack performance were analyzed by Bardou et al. [4]. In addition, they improved Bleichenbacher's attack by a factor of four. An improved attack running with the discussed oracle needs about 15,000 queries to decrypt a PKCS#1 compliant message (more queries are needed to decrypt an arbitrary message).

### 3.4 Countermeasures

Due to its importance, Bleichenbacher's attack is directly addressed in the TLS standard [11]. The basic idea of the proposed countermeasure is to continue the processing with a randomly generated `PreMasterSecret` every

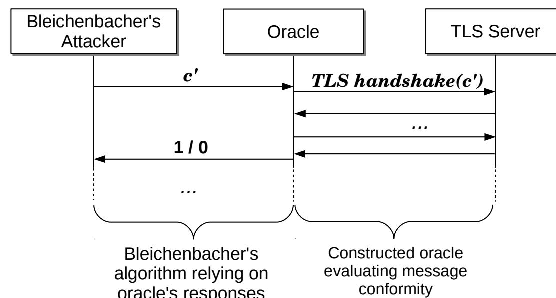


Figure 3: Bleichenbacher's attack algorithm relies on an oracle returning 1 or 0 according to the message validity.

time the message structure is invalid or decryption failed completely. This ensures unified error messages of the server. Algorithm 1 describes the implementation of this countermeasure as proposed in TLS 1.2 [11]:

**Algorithm 1** A (simplified) countermeasure against Bleichenbacher's attack proposed in the TLS standard [11].

1: generate a random $PMS_R$
2: decrypt the ciphertext: $m := dec(c)$
3: **if** ( $(m \neq 00||02||PS||00||k)$ OR $(|k| \neq 48)$ OR $(k_1||k_2 \neq maj||min)$ **then**
4: proceed with $PMS := PMS_R$
5: **else**
6: proceed with $PMS := k$
7: **end if**

This countermeasure ensures that *each* ciphertext decryption reveals a `PreMasterSecret` which is used in the handshake processing. Thus, the attacker cannot distinguish between valid and invalid ciphertexts. Note that a random `PreMasterSecret` is generated every time, independently from the ciphertext validity. This ensures equal processing times of valid and invalid ciphertexts.

## 4 SSL/TLS Penetration Testing

Given the importance of PKCS#1 format processing in SSL/TLS, it is important how Bleichenbacher countermeasures are implemented in real-world applications.

### 4.1 Attack Challenges

We investigate ways of turning a seemingly secure SSL/-TLS server into an oracle $\mathcal{O}$ suitable for Bleichenbacher's attack. The attack is sketched in Figure 3: The attacker communicates with $\mathcal{O}$ and suggests ciphertexts. $\mathcal{O}$ sends these ciphertexts to the server by performing a TLS handshake, evaluates its responses, and returns 1 or 0 according to the PKCS#1 conformity.

The oracle can be based on different side channels. First, *noisy* TLS servers responding with different error messages represent a direct oracle $\mathcal{O}_D$. Second, even if the server does not respond with different error messages, its processing logic can cause different timings while handling valid and invalid ciphertexts. These *silent* checks can be used to construct a timing oracle $\mathcal{O}_T$.

When constructing an oracle $\mathcal{O}$, we have to face the following challenges:

1. $\mathcal{O}$ must not respond with false positives: ciphertexts falsely identified as valid cause Bleichenbacher's algorithm to end up in a wrong internal state from which the algorithm cannot recover.

2. $\mathcal{O}$ should respond with as few false negatives as possible: valid ciphertexts falsely identified as invalid slow down the attack performance.

3. $\mathcal{O}$ should require as few requests as possible.

## 4.2 T.I.M.E.

This research was enabled by a new framework called T.I.M.E. - TLS Inspection Made Easy (for details see [20]). The framework implements a TLS client stack in Java with means to intercept the communication and TLS protocol flow at any time through predefined hook-points. It allows altering TLS messages in an object based representation or, if necessary, even at bit level. This renders deep analysis of TLS, simulating complex attack scenarios, or trigger bugs only occurring in usually hard to provoke operation states possible. The framework proved to be well suited for the creation of a large amount of test cases, even in complex attack scenarios. The modularity allows a quick test case creation and automated testing for vulnerabilities of many different TLS implementations with comparably little effort. A comprehensive reporting engine eases the analysis even when working with large amounts of test cases and scanning targets.

**Architecture.** Figure 4 illustrates the T.I.M.E. architecture. It consists of the following main parts:

- SSL/TLS Stack and Network Stack handle the communication between the framework and the remote SSL/TLS server.

- The Attack Engine consists of different attack modules including one for Bleichenbacher's attack. It contains the attack logic and test cases for triggering different server behavior to identify bugs in the server's SSL/TLS stack.

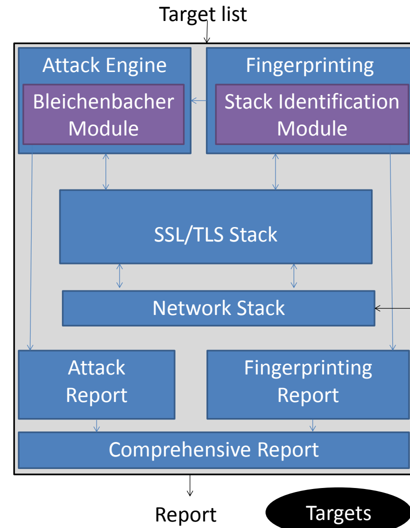


Figure 4: T.I.M.E. architecture

- The Fingerprinting Engine generates specifically formatted messages and triggers different server behavior which is analyzed to identify the SSL/TLS implementation and its version. The description of this engine is out of scope of this paper.

- The Reporting Module generates attack and fingerprinting reports.

The whole process of intercepting a running communication is event based. An application is able to register for events of interest, in this case e.g. the `ClientKeyExchange` message and `Alerts`. The workflow notifies each observer about occurring events. Once an observer is notified, the execution control is passed to this observer. The observer can manipulate the current message or internal states of the stack and return the control back to the workflow. The communication is paused until the observer returns control. Once returned the workflow continues immediately with processing.

The interaction between server, attack module and the handshake workflow of T.I.M.E. is illustrated in Figure 5.

The Bleichenbacher attack logic is built directly upon the stack and can be used to modify messages during the TLS handshake. The modified messages are used to trigger different server behavior. This allows to check for obvious vulnerabilities to Bleichenbacher's attack.

## 4.3 Test Environment

As we are performing timing attacks over a network, special care must be taken for the measurement setup. Measuring precise processing times from remote is challenging because of the jitter induced by busy network

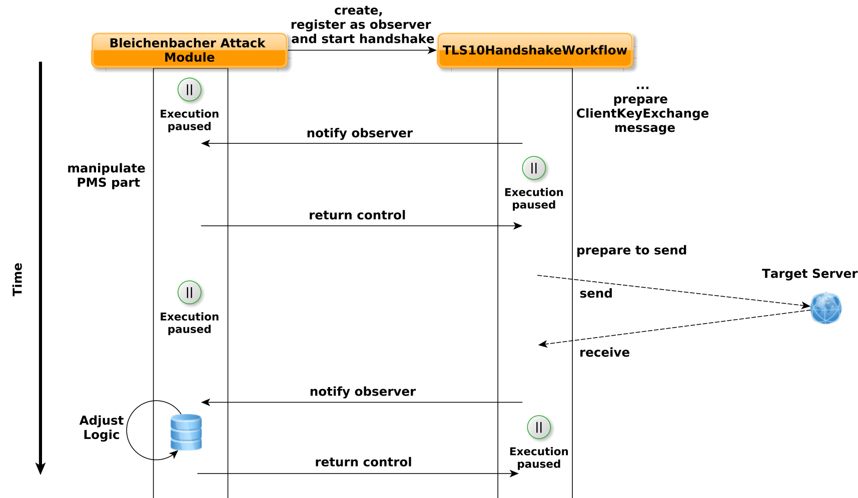


Figure 5: Interaction between the components. The Bleichenbacher Attack Module instantiates a `TLS10HandshakeWorkflow` object (part of the T.I.M.E. framework), registers as an observer for the `ClientKeyExchange` and `Alert` messages and finally starts the workflow. Every time one of these messages occurs the handshake is paused and the Bleichenbacher Attack Modules gains control. It either modifies the encrypted `PreMasterSecret` or analyzes the response message. Finally, it returns the control back to the workflow which continues with the handshake.

components, by the remote machine and by the measuring client. We also wanted to perform our attacks in a realistic scenario, in which the attacker has full control over the measuring machine, but only limited control over the network quality. We therefore ran the measurement machine with a stripped down Ubuntu 12.04 LTS Linux where we disabled CPU halting (boot parameter `idle=poll`) and CPU frequency scaling (fixing the CPU frequency using the `cpufreq` tools). Both settings are not uncommon in data centers that trade faster response times for higher power consumption. We used a Realtek 8139-based networking card with no support for interrupt coalescing. Note that this configuration likely optimizes the quality of the timing measurements, but it is not a necessary requirement. For a comprehensive analysis of hardware choices and configuration settings for timing measurements over networks see [8].

It is realistic to assume that the attacker has some limited control over the network. For example, if the connection from the attacker's machine to the target machine is of bad quality, the attacker can often rent (or compromise) a machine nearby the target machine and launch the attack from there (consider cloud-based scenarios). We therefore used a network setting in which the attacking and target machine are in the same (productive) University campus LAN connected through a Cisco Catalyst 2950 switch. This setting emulates the environment of a common co-location center or a cloud system where the attacker might even be able to rent a virtual machine that runs on the same hardware as the target machine [23].

If we use the attack module for triggering different TLS server messages, the whole T.I.M.E. tool set is placed on a single machine and communicates as a client with the remote TLS server. For timing measurement we had to act differently after we found out that T.I.M.E. provides no reliable base for highly fine grained time measurement. Thus, we decided to split the Bleichenbacher logic and the TLS logic into separate modules. Figure 6 illustrates this setup. On the left, we see the Bleichenbacher attack module that triggers and executes the attack. The Bleichenbacher logic generates new ciphertexts and hands it over to the measurement module.

To test if a TLS implementation has a suitable timing leak that allows the creation of a timing-based oracle, one has to measure the delay between the `ClientKeyExchange` message and the arrival of the `HANDSHAKE_FAILURE` message (the server performs PKCS#1 checking during this period). High precise timing measurement is not possible in Java (the JVM itself causes a significant *noise* which falsifies the results). Thus, we modified the lightweight *MatrixSSL* C implementation[5] to execute the TLS handshake and measure the timing delays in clock ticks by using the RDTSC assembler directive.

We used the timing analysis tool NetTimer[6] to evaluate the server response times. This tool implements a

[5] `http://www.matrixssl.org/`
[6] `http://sebastian-schinzel.de/nettimer`

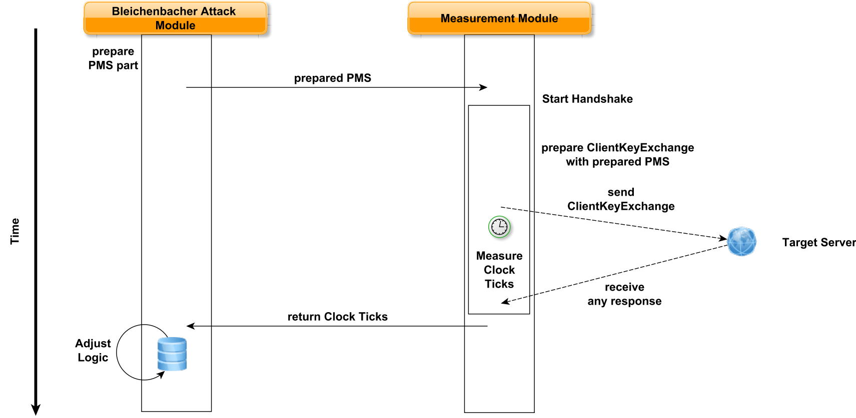


Figure 6: Architecture for measuring timing differences. The enhanced T.I.M.E. framework is split into two parts: The Bleichenbacher Attack Module and the Measurement Module based on the MatrixSSL library.

variant of Crosby's box hypothesis test, which was found to perform well for analyzing network delay measurements [8]. With this setup, we were able to reliably distinguish timing differences of a few hundred nanoseconds over a LAN with one thousand repeated measurements. This confirms that the findings of [8] not only hold for artificial UDP ping-pong protocols, but also for real-world TCP-based protocols.

## 4.4 Methodology

Our methodology during evaluation of Bleichenbacher's attack on a specific implementation can be summarized in the following steps.

**Triggering Different Server Behaviors.** In Section 3.1 we described how an encrypted ciphertext is processed on a TLS server. This process includes several validation and unpadding steps. If one of these steps is implemented incorrectly, a side channel might arise. Thus, we first implemented different T.I.M.E. test cases that aim to trigger different server behavior which could lead to a practical oracle $\mathcal{O}$. These test cases include:

1. A TLS compliant message, see Equation 2.
2. A PKCS#1 compliant message which is not TLS compliant, see Equation 1. Such a message can include a wrong TLS version number or a `PreMasterSecret` with an invalid length.
3. A non-PKCS#1 compliant message: Such a message can for example start with a non-zero byte or can be missing the `0x00` byte after the random padding of the message.

We cover all three cases and send the encrypted messages to the target TLS server and observe if the server responds with different error messages or timing behavior. As we analyze open source TLS frameworks, we are able to combine the automatic analysis of the T.I.M.E. framework with an additional source code review.

**Analyzing Oracle Strength.** We analyze if the discovered side channel can be used to construct a practical (*Strong*) Bleichenbacher oracle. This can be achieved by considering two factors. First, the probability that the oracle responds with 1 if the decrypted message starts with `0x00 02`. Second, in case of a timing oracle, how many server requests are needed to distinguish a valid from an invalid ciphertext.

**Performing the Attack.** In order to assess the practicability and performance of the attack using a constructed oracle, we use the oracle in a real attack execution and report on the number of oracle queries. For this purpose, we implemented the Bleichenbacher attack [5] as T.I.M.E. test case and extended it with the trimming and skipping holes methods from [4].

# 5 First Side Channel: Error Messages in JSSE

Automated evaluation of JSSE with T.I.M.E. revealed a new side channel which could be used to construct a noisy oracle $\mathcal{O}_{D-JSSE}$ leading to a successful Bleichenbacher attack. In general, the side channel is caused by an improper padding check and the subsequent `PreMasterSecret` processing. This behavior enabled us to force the server to respond with different alerts while processing differently formatted PKCS#1 messages: `INTERNAL_ERROR` and `HANDSHAKE_FAILURE`.

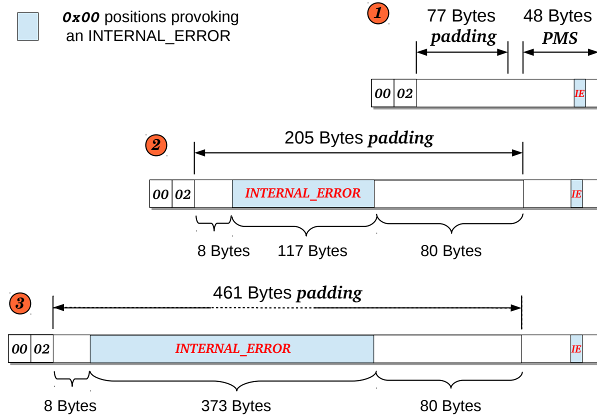


Figure 7: If a decrypted message contains a 0x00 byte preceded with non-0x00 bytes in at least one of the marked positions, JSSE responds with an INTERNAL_ERROR alert. The depicted messages are of 1024 (1), 2048 (2), and 4096 (3) bit length.

**Side Channel Analysis.** In the following, we analyze the attack on the server with a 2048 bit (256 bytes) key. Similar analysis could be applied to other key sizes.

Due to a fixed length of the PreMasterSecret (*PMS*), the padding string length can easily be determined to be 205 bytes (see Figure 2). These bytes must not include a 0x00 byte. The T.I.M.E. framework enabled us to test the JSSE implementation with specifically formatted messages. The analysis revealed that 0x00 bytes inserted at specific padding positions cause an internal ArrayIndexOutOfBoundsException leading to a different TLS alert message. The exception was caused when the PreMasterSecret length check was not correctly applied (cf. Algorithm 1, line 3). Propagation of the unchecked ArrayIndexOutOfBoundsException to the surface lead to the communication abort, the server responded with an INTERNAL_ERROR alert.

More precisely, our test revealed that changing either the first 8 or last 80 padding bytes led to a correct HANDSHAKE_FAILURE alert. Changing one of the remaining padding bytes to 0x00 caused a different INTERNAL_ERROR alert. This was caused by the MasterSecret computation initialized with a PreMasterSecret of an incorrect length. By applying 2048 bit RSA keys, the number of bytes causing an INTERNAL_ERROR alert is equal to 117 (depicted in Figure 7). In case of 4096 bit keys, this number is equal to 373 (see Figure 7).

In addition to the positions described above in 2048 and 4096 bit long ciphertexts, our analysis revealed that there is also a chance to attack 1024 bit ciphertexts directly. Independently of the applied key size, the server responded with an INTERNAL_ERROR if the second to last byte ($m_{|m|-1}$) contained 0x00 and the preceding bytes do not contain 0x00.

The different alert messages offered a new oracle $\mathcal{O}_{D-JSSE}$ responding with 1 (INTERNAL_ERROR) or 0 (HANDSHAKE_FAILURE) according to the structure of the decrypted PreMasterSecret.

**Oracle Strength.** In the following, we evaluate the probability for 2048 and 4096 bit random messages starting with 0x00 02 to contain a structure causing an INTERNAL_ERROR alert. Let $n$ be the byte size of the PKCS#1 message and $|PMS|$ the PreMasterSecret length. The number of bytes provoking an INTERNAL_ERROR can be derived as:

$$x = n - 3 - |PMS| - 8 - 80.$$

Let us consider that the first two message bytes are 0x00 02. The probability that the following 8 padding bytes are non-zero and at least one of the following $x$ bytes becomes 0x00 (and thus the server responds with an INTERNAL_ERROR alert) is:

$$P_{D-JSSE}(1|A) = \left(\frac{255}{256}\right)^8 \cdot \left(1 - \left(\frac{255}{256}\right)^x\right)$$

For key sizes of 2048 and 4096 bits (256 and 512 bytes) it results in:

$$P^{2048}_{D-JSSE}(1|A) = \left(\frac{255}{256}\right)^8 \cdot \left(1 - \left(\frac{255}{256}\right)^{117}\right) \approx 0.356$$

$$P^{4096}_{D-JSSE}(1|A) = \left(\frac{255}{256}\right)^8 \cdot \left(1 - \left(\frac{255}{256}\right)^{373}\right) \approx 0.744$$

This means that a JSSE server ($\mathcal{O}_{D-JSSE}$) using a 2048 bit RSA key responds with a probability of $P^{2048}_{D-JSSE}(1|A) \approx 35.6\%$ with 1 (INTERNAL_ERROR), if the decrypted PreMasterSecret message starts with 0x00 02. In case of using 4096 bit keys, the oracle is even more permissive. It responds with a probability of $P^{4096}_{D-JSSE}(1|A) \approx 74.4\%$ if the message starts with 0x00 02. These probabilities suggest a low number of false negatives, leading to an efficient Bleichenbacher attack.

On the other hand, when applying 1024 bit long RSA keys, $\mathcal{O}_{D-JSSE}$ is much less permissive. It responds with an INTERNAL_ERROR only if 0x00 is positioned just before the last byte. Thus, the probability $P^{1024}_{D-JSSE}(1|A)$ can be computed as:

$$P^{1024}_{D-JSSE}(1|A) = \left(\frac{255}{256}\right)^{124} \cdot \left(\frac{1}{256}\right) \approx 0.0024$$

|  | Mean | Median |
|---|---|---|
| 2048 bit RSA key | 176,797 | 37,399 |
| 4096 bit RSA key | 73,710 | 27,744 |

Table 2: Number of required queries to execute an optimized Bleichenbacher's attack on a JSSE server using 2048 bit and 4096 bit RSA keys.

**Attack Evaluation.** We used this oracle to perform a Bleichenbacher attack – the experiment was repeated 1,000 times. Results of this evaluation confirm the findings of our theoretical analysis from the previous section: Executing the attack using a less restrictive oracle with a 4096 bit RSA key leads to fewer oracle queries. We needed about 177,000 queries to a JSSE server applying 2048 bit keys and about 74,000 queries to a JSSE server applying 4096 bit keys. See Table 2 for details.

We performed full `PreMasterSecret` recovery attacks against a TLS server working with 2048 bit keys. With our T.I.M.E. framework we were able to send about 3.85 server queries per second. Thus, sending 177,000 requests lasted about 12 hours. The attack was performed on localhost.[7]

Performance evaluation of an oracle using 1024 bit keys resulted in hundreds of millions of oracle queries. This is caused by the high restrictiveness of $\mathcal{O}_{D-JSSE}$ when applying keys of this length.

**Mitigation.** We communicated this problem to the Oracle Security response team and the bug was assigned CVE-2012-5081. The attack is fixed with the *Oracle Java SE Critical Patch October 2012 – Java SE Development Kit 6, Update 37 (JDK 6u37).*

## 6 Second Side Channel: Timing Differences in OpenSSL

The discovery of the aforementioned vulnerability in JSSE motivated to investigate the source code of open source SSL/TLS frameworks. We reviewed JSSE, GnuTLS and OpenSSL and found that they do not implement the countermeasure against Bleichenbacher's attack as proposed by the TLS 1.2 specification [11].

**Side Channel Analysis.** The countermeasure against this attack is mostly implemented as depicted in Algorithm 1. The important observation is that the random key is generated if, and only if, the received key is not

[7] Improving the T.I.M.E. sending performance would result in much faster attack executions. This was however not the primary goal of our work.

**Algorithm 2** Improper implementation of the countermeasure against Bleichenbacher's attack (suggested by TLS 1.0 and TLS 1.1) possibly causing a timing side channel in all the analyzed implementations.

1: decrypt the ciphertext: $m := dec(c)$
2: **if** ( $(m \neq 00||02||PS||00||k)$ OR $(|k| \neq 48)$
    OR $(k_1||k_2 \neq maj||min)$ **then**
3:     generate a random $PMS_R$
4:     proceed with $PMS := PMS_R$
5: **else**
6:     proceed with $PMS := k$
7: **end if**

TLS compliant (see Equation 2). Thus, the random key generation and the assignment create a new timing side channel that leaks information about the TLS compliance of a received `PreMasterSecret`. These processing steps have independently been observed and criticized by Matthew Green [18].

**Oracle Strength.** *Timing Reliance.* We tested the timing differences between valid and invalid ciphertexts with OpenSSL 0.98. Figure 8 shows the filtered results of our timing analysis over a LAN with 5,000 measurements. The results suggest that we can distinguish TLS compliant and non-PKCS#1 compliant ciphertexts. We could achieve similar results for OpenSSL 1.01.

Even though the results clearly showed constant differences of about 1.5 microseconds, we are not sure if the root cause of these differences is additional random number generation. The OpenSSL code contained sev-

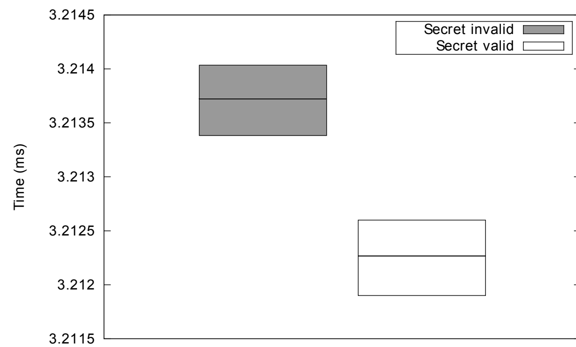


Figure 8: Timing measurement results for OpenSSL 0.98. The valid secret refers to a TLS compliant ciphertext. The invalid secret refers to a non-PKCS#1 compliant ciphertext. In the non-PKCS#1 compliant structure the first byte (which should be `0x00`) was altered to `0x08` to provoke a random number generation on the TLS server.

eral additional branches and loops in the PKCS#1 processing which could blur our results. The analysis of this problem showed up to be very difficult and related to compile flags. Despite this uncertainty, our measurements clearly show that a side channel exists.

*Probability Analysis.* The analyzed timing behavior can be used to construct an oracle

$$\mathcal{O}_{T-rand}(c) = \begin{cases} 1 & \text{TLS compliant} \\ 0 & \text{non-TLS compliant (with an additional random number generation)} \end{cases}$$

However, it does not lead to a practical attack. An oracle created from this timing leak is very "weak". It responds to an oracle request with 1 if, and only if, the decrypted ciphertext is TLS compliant (see Equation 2). For a 2048 bit key, the probability that an oracle responds with 1 in case that the decrypted message starts with `0x00 02` is very low:

$$P^{2048}_{T-rand}(1|A) = \left(\frac{255}{256}\right)^{205} \cdot \left(\frac{1}{256}\right)^{3} \approx 2.7 \cdot 10^{-8}$$

The reason is that 205 padding bytes must be non-zero and the following bytes must contain `0x00`$||maj||min$. See Figure 2.

**Attack Evaluation.** $\mathcal{O}_{T-rand}$ is very "weak" and did not allow to execute a practical Bleichenbacher attack. We were only able to estimate the number of oracle queries. According to Bleichenbacher and Bardou et al. [5, 4], the number of oracle queries for the complete attack can be computed as:

$$(2^{17} + 16 \cdot 256)/P_{T-rand} = 5 \cdot 10^{12}$$

**Mitigation.** The mitigation is described in RFC 5246 [11]. Algorithm 1 illustrates the correct processing: A random value should always be generated, *before* processing the decrypted data.

## 7 Third Side Channel: Internal Exception

We decided to search for different side channels leading to more practical oracles. As pointed out by James Manger on the official JOSE (JSON Object Signing and Encryption) mailing list,[8] an additional side channel could arise from an improper `Exception` handling in Java's PKCS#1 implementation.

[8] `http://www.ietf.org/mail-archive/web/jose/current/msg01936.html`

**Side Channel Analysis.** The Java PKCS#1 implementation strictly checks the message format according to Equation 1. The message must start with `0x00 02`, contain at least eight non-zero padding bytes, and a `0x00` byte indicating the end of the padding string. If this format is correct, the secret is extracted. Otherwise, a `BadPaddingException` is thrown. The method code can be found in Listing 9.

```
1  /**
2   * PKCS#1 v1.5 unpadding (blocktype 1 and 2).
3   */
4  private byte[] unpadV15(byte[] padded)
5   throws BadPaddingException {
6     int k = 0;
7     if (padded[k++] != 0) {
8       throw new BadPaddingException(
9        "Data must start with zero");
10    }
11    if (padded[k++] != type) {
12      throw new BadPaddingException(
13       "Blocktype mismatch: " + padded[1]);
14    }
15    while (true) {
16      int b = padded[k++] & 0xff;
17      if (b == 0) {
18        break;
19      }
20      if (k == padded.length) {
21        throw new BadPaddingException(
22         "Padding string not terminated");
23      }
24      if ((type == PAD_BLOCKTYPE_1)
25       && (b != 0xff)) {
26        throw new BadPaddingException(
27         "Padding byte not 0xff: " + b);
28      }
29    }
30    int n = padded.length - k;
31    if (n > maxDataSize) {
32      throw new BadPaddingException(
33       "Padding string too short");
34    }
35    byte[] data = new byte[n];
36    System.arraycopy(padded,
37     padded.length - n, data, 0, n);
38    return data;
39  }
```

Figure 9: Java's PKCS# v1.5 method for format check and padding removal can throw a BadPaddingException - *Source: sun.security.rsa.RSAPadding*

With our T.I.M.E. framework we investigated the JSSE server implementation which internally uses the PKCS#1 unpadding method described above. We sent PKCS#1 compliant and non-PKCS#1 compliant messages to the JSSE server and found that with non-PKCS#1 compliant messages an *additional* `Exception` could be provoked. The `Exception` was correctly handled by the JSSE logic and did not result in a distinguish-

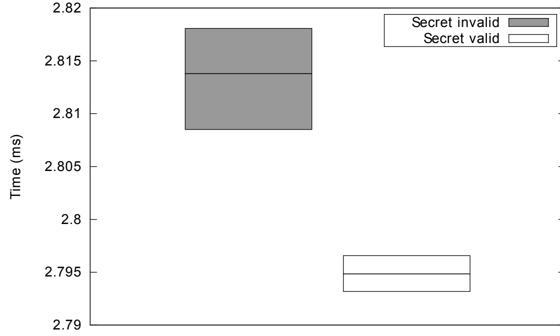


Figure 10: Timing measurement results for Java 1.7 (JSSE). The valid secret refers to a PKCS#1 compliant ciphertext. The invalid secret refers to a non-PKCS#1 compliant ciphertext. In the non-PKCS#1 compliant structure the first byte (which should be 0x00) was altered to 0x08 to provoke an exception on the TLS server.

able error message. Thus, it did not help to create a direct PKCS#1 validation oracle. However, `Exception` handling in Java (as well as in other object oriented languages) can introduce timing delays and thus slow down the whole application. Throwing, catching, and handling an `Exception` are time consuming tasks and thus lead to additional processing time.

**Oracle Strength.** *Timing Reliance.* We analyzed the timing differences between processing PKCS#1 compliant and non-PKCS#1 compliant messages on TLS servers running on Java 1.6 and 1.7 platforms. Figure 10 shows the filtered results of our time measurement with 5,000 queries. The results show differences of about 20 microseconds.

*Probability Analysis.* This behavior allows us to construct a new timing oracle:

$$\mathcal{O}_{T-exc}(c) = \begin{cases} 1 & \text{PKCS\#1 compliant} \\ 0 & \text{non-PKCS\#1 compliant (with an additional internal exception handling)} \end{cases}$$

$\mathcal{O}_{T-exc}$ is very permissive and much stronger than $\mathcal{O}_{T-rand}$, because it contains fewer plaintext validity checks. When working with 2048-bit keys, this oracle responds to a request starting with 0x00 02 with 1 with the following probability:

$$P^{2048}_{T-exc}(1|A) = \left(\frac{255}{256}\right)^{8} \cdot \left(1 - \left(\frac{255}{256}\right)^{246}\right) \approx 0.6$$

Applying such an oracle results in much lesser queries and can thus be expected to be used for a practical attack.

**Attack Evaluation.** We used this timing oracle $\mathcal{O}_{T-exc}$ to perform a real Bleichenbacher attack in a switched LAN and proved the practicability of $\mathcal{O}_{T-exc}$. The attack on OpenJDK 1.6 took about 19.5 hours and 18,600 oracle queries.[9] About 20% of PKCS#1 compliant messages were identified as non-PKCS#1 compliant. The attack on Java 1.7 took about 55 hours and 20,662 queries. The larger number of queries and the longer processing time are caused by a higher value of false negatives (about 50%). The oracle identified about 467 PKCS#1 compliant messages incorrectly.

**Mitigation.** The object oriented architecture and especially the `Exception` handling of the JSSE implementation makes fixing the timing leak challenging. A common implementation pattern for RSA decryption is to provide a (generic) function to which the ciphertext is passed which returns the plaintext on success or an `Exception` otherwise. As stated, the generation of the `Exception` creates a detectable timing difference. Preparing an `Exception` at function entry, but not throwing it, leads to a smaller time difference, but might still be exploitable.

As a consequence we implemented a time constant PKCS#1 processing for SSL/TLS and proposed it as a fix for this issue to Oracle. The bug was assigned CVE-2014-411 and it was fixed with the *Oracle Java SE Critical Patch January 2014 – Java SE 7, Update 45* (and with the previous versions *Java SE 5u55* and *6u65*).

We verified that a similar timing behavior based on an additional exception is observable in a widespread BouncyCastle library.[10] BouncyCastle is implemented in two languages: Java and C#. We tested both implementations and locally invoked BouncyCastle PKCS#1 decryption methods. We could observe timing differences of about 20 microseconds between valid and invalid PKCS#1 messages in the Java and C# BouncyCastle version. This proved that the described timing behavior is not Java specific, and can be found in other object-oriented languages as well. We developed a patch for the Java version of BouncyCastle. We contacted the BouncyCastle developers with the proposed patch in March 2014.

## 8 Fourth Side Channel: Unexpected Timing Behavior by Hardware Appliances

The performance and practicability of the previous attacks motivated us to analyze further TLS stacks. We

[9] One oracle query is not equal to one server request. In order to respond to an oracle query, $\mathcal{O}_{T-exc}$ issued in our scenario up to 750 real server requests. It evaluated the response times and decided if the ciphertext was valid or not. See Figure 3.

[10] `https://www.bouncycastle.org`

had a chance to evaluate the behavior of *F5 BIG-IP* and *IBM Datapower* which use the *Cavium NITROX SSL accelerator chip*. Automated evaluation with our T.I.M.E. framework revealed that it was possible to execute a *complete* handshake, even though the encoded `PreMasterSecret` was of an incorrect format. More precisely, *F5 BIG-IP* and *IBM Datapower* did not verify the first byte of the PKCS#1 message and accepted messages which started with `0x??02` (where `0x??` represents an arbitrary byte).

**Side Channel Analysis.** This behavior does not lead to a direct attack. In order to correctly complete a handshake flow and receive a *Server*`Finished` message, an authenticated *Client*`Finished` message has to be sent to the server. Otherwise, the analyzed server responds with a `HANDSHAKE_FAILURE` message. Since the Bleichenbacher attacker is not in possession of the `PreMasterSecret`, he is not able to authenticate the *Client*`Finished` message and thus cannot trigger different messages. However, the server behavior strongly indicated that there could be a leakage in the PKCS#1 processing. Even though this leakage did not lead to different server responses, we assumed we could observe timing differences.

In comparison to the analysis described in the previous sections, we had no chance to review the code, because it is not publicly available. This turned our work to a black-box analysis and made it much harder.

**Oracle Strength.** We had a chance to evaluate the timing behavior of an *IBM Datapower* directly in our lab. The measurement machine was connected with a router to the *IBM Datapower* appliance.[11] We created different TLS requests based on our methodology (TLS compliant requests, PKCS#1 compliant requests, invalid requests etc.), and sent these requests to the server while the measurement machine observed the response times. The response times were finally compared using the NetTimer library.

The comparison of the response times confirmed our predictions and we could see clear timing differences by processing our TLS requests. The most visible timing difference was produced by requests starting with `0x?? 02`, see Figure 11. Based on this timing difference, the server behavior allowed to construct a new timing oracle:

$$\mathcal{O}_{T-hard} = \begin{cases} 1 & \text{starts with } \texttt{0x??02} \\ 0 & otherwise \end{cases}$$

[11] In comparison to the previous measurements, the router did not route real traffic so our experiments were executed in a "lab" scenario.

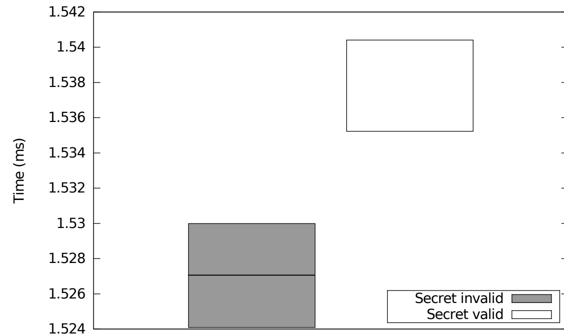


Figure 11: Timing measurement results for our *IBM Datapower*. The valid secret refers to a message, which starts with `0x??02`, where `0x??` indicates an arbitrary byte. The invalid secret refers to a message starting with different bytes.

However, this oracle is not compliant to the oracle used by Bleichenbacher. It responds with 1 to the request starting with `0x0102`, `0x0202`, `0x0302`, .... Thus, we needed to modify and adapt the original algorithm to handle this special case. This novel variant is described in Section 9.

**Attack Evaluation.** We evaluated the performance of our algorithm using a test oracle behaving like $\mathcal{O}_{T-hard}$. We repeated our experiment 500 times, with a 2048 bit RSA key. We needed about 4700 queries (median) to decrypt a ciphertext. This high performance is caused by the higher number of intervals the oracle accepts. Manger's attack [19] also reveals similar behavior.

We used the constructed timing oracle $\mathcal{O}_{T-hard}$ to perform a real attack on an *IBM Datapower* appliance. Our attacker needed 7371 oracle queries. The oracle correctly evaluated 2033 valid ciphertexts, while 1290 valid ciphertexts were incorrectly evaluated as invalid. The attack lasted 41 hours. The timing oracle $\mathcal{O}_{T-hard}$ issued about 4,000,000 server queries in total.

**Mitigation.** We communicated our findings to the vendors in November 2013. The current state of these issues can be tracked on their websites. F5 tracks this problem in their Bugzilla database under ID 435652. IBM gives their customers information about the current state in the Security Bulletin: SSL/TLS side channel attack on WebSphere DataPower (CVE-2014-0852).[12]

Since the Cavium products are used by other vendors like Cisco, Citrix or Juniper Networks, we assume that many other products were vulnerable, too.[13]

[12] `http://www.ibm.com/support/docview.wss?uid=swg21678204`

[13] `http://www.cavium.com/winning_products.html`

## 9 Novel Bleichenbacher Attack Variant

In the previous section we described a new oracle $\mathcal{O}_{T-hard}$. The oracle responds with 1 if a decrypted message starts with 0x??02, where 0x?? represents an arbitrary byte. Such an oracle is not strong enough to implement Bleichenbacher's attack. The original algorithm from [5] is not able to tolerate false positives, it requires an oracle responding with 1 *only if* the decrypted message starts with 0x0002. Note that $\mathcal{O}_{T-hard}$ is much weaker, as it responds with 1 if the message starts with 0x??02. In the following we describe a novel variant of Bleichenbacher's attack, which is more robust than the original one and works also with the weaker oracle $\mathcal{O}_{T-hard}$.

We assume that the original message is PKCS#1 compliant and lies in the interval $[2B, 3B)$, where $B = 2^{8(\ell-2)}$. In this case the Bleichenbacher algorithm sets the starting interval containing the message of interest $m_0 \in [a, b]$, where $a = 2B$ and $b = 3B$

In the first step, the original algorithm searches for values $s > (2B+N)/3B$ such that $c = (c_0 \cdot s^e) \bmod N$ is decrypted to a PKCS#1 compliant message. This is not possible by applying $\mathcal{O}_{T-hard}$, since the oracle would respond with many false positives. We know that if $\mathcal{O}_{T-hard}$ responds with 1, the decrypted message starts with 0x0002, 0x0102, ... or 0xFF02. This means the message lies in one of the following intervals: $[2B, 3B)$, $[258B, 259B)$, $[514B, 515B)$, .... If we start the algorithm with a large $s$ value, we can easily produce a message from one of those intervals.

The basic idea behind our algorithm is to use the additional intervals and make the search more fine-grained. For this purpose, we define $q$, where $q \in \{1 \dots (N/256B)\}$. In the first step, we set $r_0 = 0$ and iteratively search $s_{ij}$ values by setting $q_j = 1 \dots (N/256B)$:

$$\frac{2B + r_i N + q_j(256B)}{b} \leq s_{ij} < \frac{3B + r_i N + q_j(256B)}{a}.$$

We send $(c_0 \cdot s_{ij}^e) \bmod N$ to the server and observe its response. With each valid request, we can reduce the interval, where the original plaintext $m_0$ lies in:

$$a = max\left(a, \frac{2B + r_i N + q_j(256B)}{s_{ij}}\right)$$

$$b = min\left(b, \frac{3B + r_i N + q_j(256B)}{s_{ij}}\right)$$

Afterwards, we increment $r$ and repeat the same steps for $q = 1 \dots (N/256B)$.

The algorithm repeats these steps and reduces the possible solutions for $m_0$, until only one solution is left.

## 10 Other TLS Stacks

During our research we also analyzed other SSL/TLS implementations. *Microsoft Schannel* (Secure Channel) revealed no significant timing differences and behaves quite differently to any other stack: In case of processing errors of any kind, the connection is immediately terminated instead of sending alert messages. The timing measurements were too noisy to distill boundaries for distinguishing different processing branches. Due to the fact that the product is closed-source a code analysis was not possible.

## 11 Related Work

In this section we give a short overview on scientific publications analyzing side channel attacks and security of SSL/TLS. For a comprehensive list of SSL/TLS attacks we refer to [21].

**Bleichenbacher Attacks.** After publication of the original attack [5], several variants were discovered. Klima et al. found out that a strict verification of the TLS version number in the PreMasterSecret can lead to a side channel enabling Bleichenbacher's attack [16]. In [4] Bardou, Focardi, Kawamoto, Simionato, Steel and Tsay significantly improved Bleichenbacher's attack, and applied it to other PKCS#1-based environments.

Although Daniel Bleichenbacher conjectured that there might be timing-based side channels for Bleichenbacher attacks, they were discovered only for other protocols. For example, Jager et al. [13] describe a practical timing-based Bleichenbacher attack against implementations of the XML Encryption standard. They were able to exploit this side channel over a very noisy network (Planetlab) which was possible because timing differences could be increased by the attacker. During their research, they measured timing differences in the order of milliseconds whereas we had to cope with microseconds.

**Timing Attacks on SSL/TLS.** In 2003, Brumley and Boneh described an attack based on a timing side channel SSL/TLS [7], applicable if RSA is used for key exchange. Based on timing differences during processing of specially crafted ClientKeyExchange messages the private key of a server could successfully be extracted. Additionally, in 2011 Brumley and Tuveri [6] successfully attacked ECDSA based TLS connections (only OpenSSL stacks) by exploiting performance tweaks of the implementation.

**Recent Attacks on SSL/TLS.** The BEAST attack by Rizzo and Duong exploits predictable initialization vectors used by AES-CBC in TLS 1.0 [24]. The CRIME attack of the same authors shows that application of a compression method on plaintexts transported over SSL/TLS can lead to serious practical attacks. Both attacks were theoretically discussed before [3, 15]. The authors showed how to apply them practically in specific scenarios by exploiting additional side channels. AlFardan and Paterson presented the Lucky13 padding oracle attack on AES-CBC [2] which exploits timing differences revealed by the HMAC computation over the decrypted data.

To practically deploy these attacks, a strong attacker is needed who is able to force the victim to *repeatedly* send the *same* data to the server. In contrast, our attacks exploit new side channels to mount Bleichenbacher's attack which enables to decrypt the whole `PreMasterSecret` (and thus the whole SSL/TLS session) without the need to control the user's client software.

**Theoretical Results on TLS Security.** After publication of Bleichenbacher's paper, the security of encoding schemes for RSA-based TLS was discussed intensively. However, due to the fact that the `Finished` messages are sent encrypted, no full security proof for TLS was available prior to 2012. In [12], a new security model (ACCE) was introduced by Jager et al., and a full proof for TLS-DHE with mutual authentication was given.

One year later, Krawczyk et al. gave a proof for the two remaining families of ciphersuites, TLS-RSA and TLS-DH, and for server-only authentication [17]. They prove security against Bleichenbacher attacks by proposing the following countermeasure: The server should use the `ClientFinished` message as a Message Authentication Code (MAC) for the `ClientKeyExchange` message. Only if `ClientFinished` is verified successfully, the server should continue the handshake by making further computations.

These two papers contain extensive related work sections, where all previous theoretical publications on TLS can be found. Theoretical security proofs must be treated carefully: The results can only be applied to practical implementations if all preconditions are satisfied, and if all cryptographic building blocks are implemented in an ideal way (i.e. yielding no side channels). Our results thus do not contradict the proofs, but simply show that the implementations of the building blocks are not ideal.

## 12 Future Work

**TLS for non-HTTP protocols.** The search for new error or timing-based side channels can be broadened to cover cryptographic protocol implementations in other application scenarios. Especially, protocols that use parts or concepts of SSL/TLS, such as EAP-TLS [1] or SSL/TLS stacks of other languages and frameworks provide space for further investigation.

```
1  /**
2   * PKCS#1 v2.1 OAEP unpadding (MGF1).
3   */
4  private byte[] unpadOAEP(byte[] padded)
5   throws BadPaddingException {
6     byte[] EM = padded;
7     int hLen = lHash.length;
8
9     if (EM[0] != 0) {
10        throw new BadPaddingException(
11          "Data must start with zero");
12    }
13    ...
```

Figure 12: OAEP unpadding function of Java 7.

**OAEP Comes to the Rescue.** Many problems related to the *old* PKCS#1 are supposed to disappear with the introduction of OAEP [14]. However, during our research we also found problems in Java's OAEP processing. Listing 12 shows the code of Java's `RSAPadding.java` class which contains the logic for OAEP processing.

Line 9-12 outline a conditional branch that could be used to apply Manger's attack [19]. Patching is required. This example shows that OAEP is only of help if implemented correctly, i.e. without side channels.

We notified Oracle about this issue. The code was patched in the Java release from April 2014.

## 13 Conclusion

The problem of side channels leaking partial information about cryptographic computations seems to be much more persistent than expected: Error messages from standard libraries, and especially timig issues make generic solutions impossible.

The results of this paper show that Bleichenbacher attacks can still be used to break SSL/TLS implementations. Timing side channels underline the need for cryptographic libraries with branch independent, nearly time constant execution paths. The uncovered side channels motivate for the development of cryptographic penetration testing tools, able to detect such implementation deficiencies in the development phase.

Our results are alarming, especially when considering that Bleichenbacher attacks are known for about 15 years. They also show that PKCS#1 compliance checking is of prime importance to the security of a TLS implementations: Strict checks on TLS-PKCS#1 compli-

ance as performed by OpenSSL prevent Bleichenbacher attacks, even if side channels are present.

The question whether the introduction of RSA-OAEP padding would solve the problem still remains open: Only if RSA-OAEP is implemented without any side channels, the cryptographic features of this padding scheme can be enforced.


## Acknowledgements

We would like to thank Graham Steel for providing us their improved Bleichenbacher attack code [4], and the security teams of Oracle, Cavium, IBM and F5 for their cooperation.

Furthermore, we would like to thank Tibor Jager, Christian Mainka, James Manger, and anonymous reviewers for their comments.


## References


[1] ABOBA, B., BLUNK, L., VOLLBRECHT, J., CARLSON, J., AND LEVKOWETZ, H. Extensible Authentication Protocol (EAP). RFC 3748 (Proposed Standard), June 2004. Updated by RFC 5247.

[2] ALFARDAN, N. J., AND PATERSON, K. G. Lucky thirteen: Breaking the tls and dtls record protocols. *2013 IEEE Symposium on Security and Privacy 0* (2013), 526–540. `http://www.isg.rhul.ac.uk/tls/TLStiming.pdf`.

[3] BARD, G. V. The vulnerability of ssl to chosen plaintext attack. *IACR Cryptology ePrint Archive 2004* (May 2004), 111.

[4] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., STEEL, G., AND TSAY, J.-K. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology – CRYPTO* (2012), Canetti and R. Safavi-Naini, Eds.

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

[6] BRUMLEY, B., AND TUVERI, N. Remote Timing Attacks Are Still Practical. In *Computer Security - ESORICS 2011*, vol. 6879 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Sept. 2011.

[7] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (June 2003), SSYM'03, USENIX Association.

[8] CROSBY, S. A., WALLACH, D. S., AND RIEDI, R. H. Opportunities and limits of remote timing attacks. *ACM Trans. Inf. Syst. Secur. 12*, 3 (Jan. 2009), 17:1–17:29.

[9] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746.

[10] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), Apr. 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746.

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

[12] JAGER, T., KOHLAR, F., SCHÄGE, S., AND SCHWENK, J. On the security of tls-dhe in the standard model. In *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 273–293.

[13] JAGER, T., SCHINZEL, S., AND SOMOROVSKY, J. Bleichenbacher's attack strikes again: Breaking pkcs#1 v1.5 in xml encryption. In *ESORICS* (2012), S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science*, Springer, pp. 752–769.

[14] JONSSON, J., AND KALISKI, B. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), Feb. 2003.

[15] KELSEY, J. Compression and information leakage of plaintext. In *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers* (Nov. 2002), vol. 2365 of *Lecture Notes in Computer Science*, Springer.

[16] KLÍMA, V., POKORNÝ, O., AND ROSA, T. Attacking RSA-Based Sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, vol. 2779 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Sept. 2003.

[17] KRAWCZYK, H., PATERSON, K. G., AND WEE, H. On the Security of the TLS Protocol: A Systematic Analysis. Cryptology ePrint Archive, Report 2013/339, 2013. `http://eprint.iacr.org/`.

[18] M. D. GREEN (@OPENSSLFACT). OpenSSL vs. best practices (RSA decryption edition). 2.10.2012. 16:04, Tweet, `https://twitter.com/OpenSSLFact/status/253060773218222081`.

[19] MANGER, J. A chosen ciphertext attack on rsa optimal asymmetric encryption padding (oaep) as standardized in pkcs #1 v2.0. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings* (2001), vol. 2139 of *Lecture Notes in Computer Science*, Springer, pp. 230–238.

[20] MEYER, C. *20 Years of SSL/TLS Research : An Analysis of the Internet's Security Foundation*. PhD thesis, Ruhr-University Bochum, Feb. 2014.

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

[22] PATERSON, K. G., RISTENPART, T., AND SHRIMPTON, T. Tag size does matter: attacks and proofs for the TLS record protocol. In *Proceedings of the 17th international conference on The Theory and Application of Cryptology and Information Security* (Dec. 2011), ASIACRYPT'11, Springer-Verlag.

[23] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.

[24] RIZZO, J., AND DUONG, T. Here Come The XOR Ninjas, May 2011.