Fachhochschule
Münster University of
Applied Sciences

Revisiting SSL/TLS implementations
31c3

Sebastian Schinzel

Email: schinzel (a) fh-muenster (.) de
Web: https://www.its.fh-muenster.de/
Twitter: @seecurity

Background

- Sebastian: cs Professor for information security at Münster University of Applied Sciences
- Former talks at CCC:
  - 28c3: Time is on my side
  - 29c3: Time is not on your side

- This Talk is based on academic paper:

“Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks”
Meyer, Somorovsky, Weiss, Schwenk, Schinzel, Tews.
23rd Usenix Security Symposium 2014.

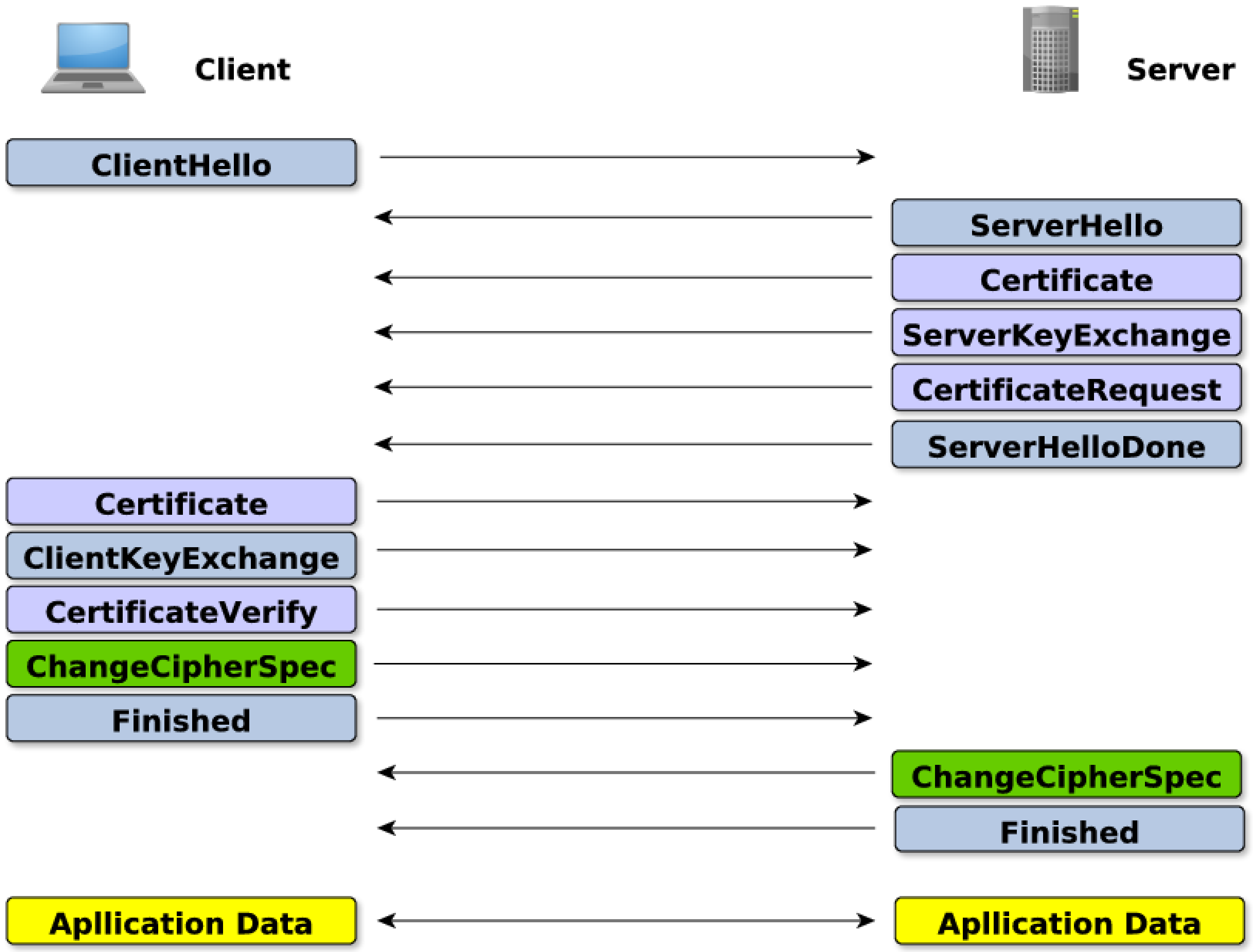
Background

Lots and lots of SSL/TLS bugs in the last few years

- Recently: Heartbleed, goto fail, POODLE, CRIME, BEAST, BREACH, Lucky 13, RC4 bias, Triple Handshake attack, ...
- >10 years ago: Bleichenbacher attack, Brumley-Boneh attack, …
- Some were protocol-level bugs, some were implementation-level bugs
  → Designing crypto protocols is hard
  → Implementing crypto protocols is hard
- Some protocol-level decisions lead to fragile implementations

Revisting SSL/TLS implementations

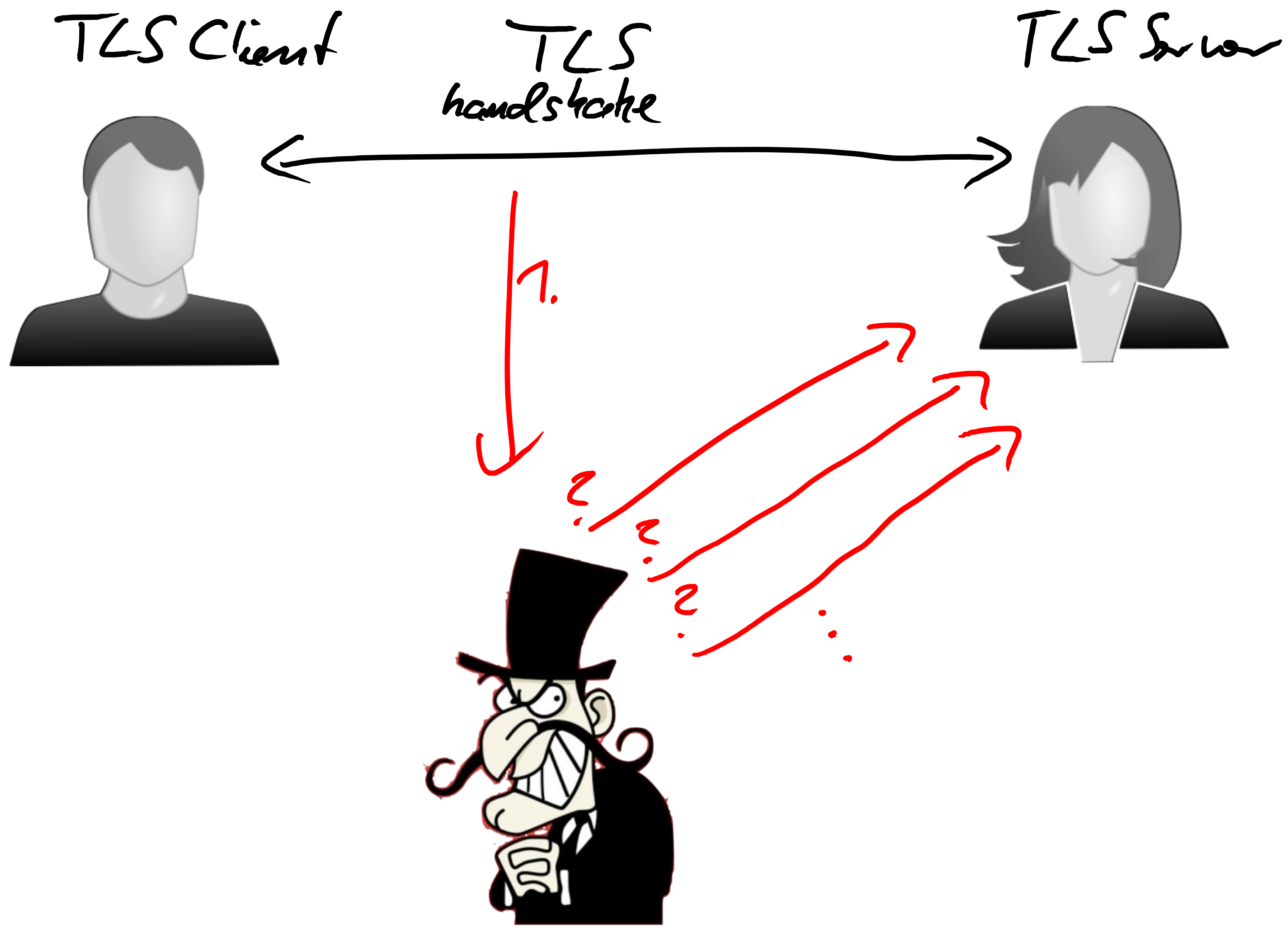
- Hybrid crypto in TLS:
  - symmetric encryption for actual TLS payload
  - asymmetric encryption for exchanging the symmetric "MasterSecret"
- Client generates random PreMasterSecret (PMS)
- Client encrypts PMS with server's public key and sends it so server
- MasterSecret is derived from PMS



Attacker Scenario

Attacker can decrypt PreMasterSecret using an adaptive chosen ciphertext attack

1. Attacker records encrypted TLS handshake
2. Attacker decrypts PreMasterSecret of that handshake by sending many modified cipher texts to the server and watching the server's behavior



Fachhochschule Münster University of Applied Sciences

Our attack works against flawed implementations of RSA-based TLS cipher suites

→no ECC suites

→no Diffie-Hellman suites

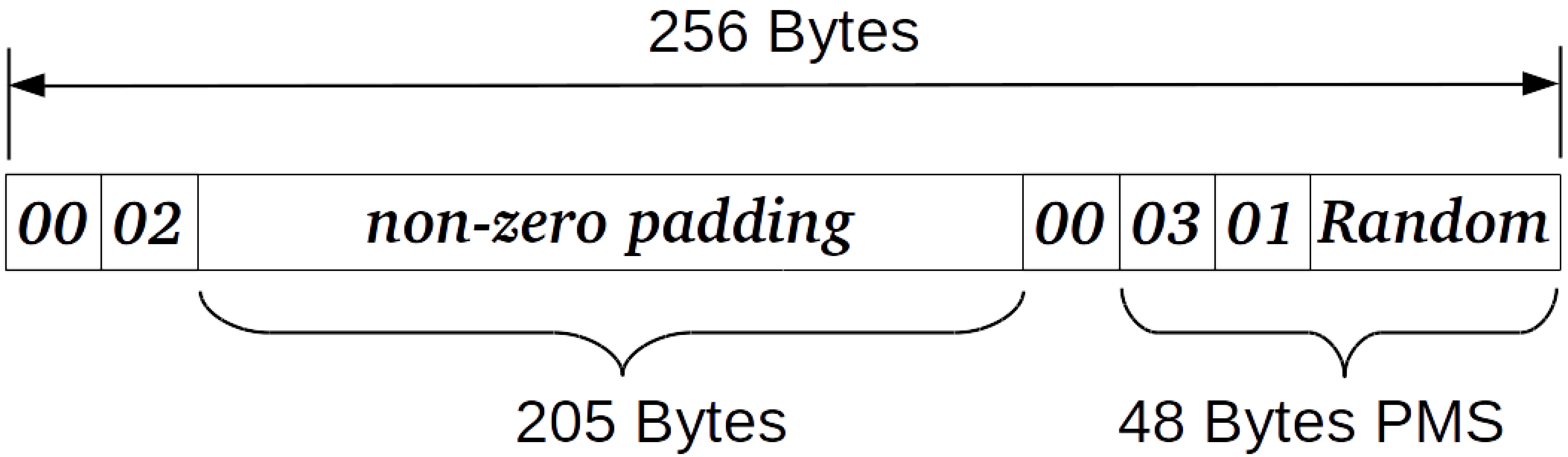
RFC4162	TLS_RSA_WITH_SEED_CBC_SHA
RFC4346	TLS_RSA_EXPORT_WITH_RC4_40_MD5
RFC4346	TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
RFC4346	TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
RFC5246	TLS_RSA_WITH_RC4_128_MD5
RFC5246	TLS_RSA_WITH_RC4_128_SHA
RFC5246	TLS_RSA_WITH_3DES_EDE_CBC_SHA
RFC5246	TLS_RSA_WITH_AES_128_CBC_SHA
RFC5246	TLS_RSA_WITH_AES_128_CBC_SHA256
RFC5246	TLS_RSA_WITH_AES_256_CBC_SHA
RFC5246	TLS_RSA_WITH_AES_256_CBC_SHA256
RFC5288	TLS_RSA_WITH_AES_128_GCM_SHA256
RFC5288	TLS_RSA_WITH_AES_256_GCM_SHA384
RFC5469	TLS_RSA_WITH_DES_CBC_SHA
RFC5469	TLS_RSA_WITH_IDEA_CBC_SHA
RFC5932	TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
RFC5932	TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
RFC5932	TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
RFC5932	TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
RFC6209	TLS_RSA_WITH_ARIA_128_CBC_SHA256
RFC6209	TLS_RSA_WITH_ARIA_128_GCM_SHA256
RFC6209	TLS_RSA_WITH_ARIA_256_CBC_SHA384
RFC6209	TLS_RSA_WITH_ARIA_256_GCM_SHA384
RFC6367	TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256
RFC6367	TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384
RFC6655	TLS_RSA_WITH_AES_128_CCM
RFC6655	TLS_RSA_WITH_AES_128_CCM_8
RFC6655	TLS_RSA_WITH_AES_256_CCM
RFC6655	TLS_RSA_WITH_AES_256_CCM_8

The RSA encryption algorithm

- Encryption: $c = m^e \ (mod\ n)$
- Decryption: $m = c^d \ (mod\ n)$
- RSA is malleable: changes in ciphertexts have predictable effects on cleartext

$$c = (c_0\ s^e)\ mod\ n = (m_0\ s)^e\ mod\ n$$

- PMS uses padding defined by PKCS#1 v1.5
- Example for a 2048 bit public key:

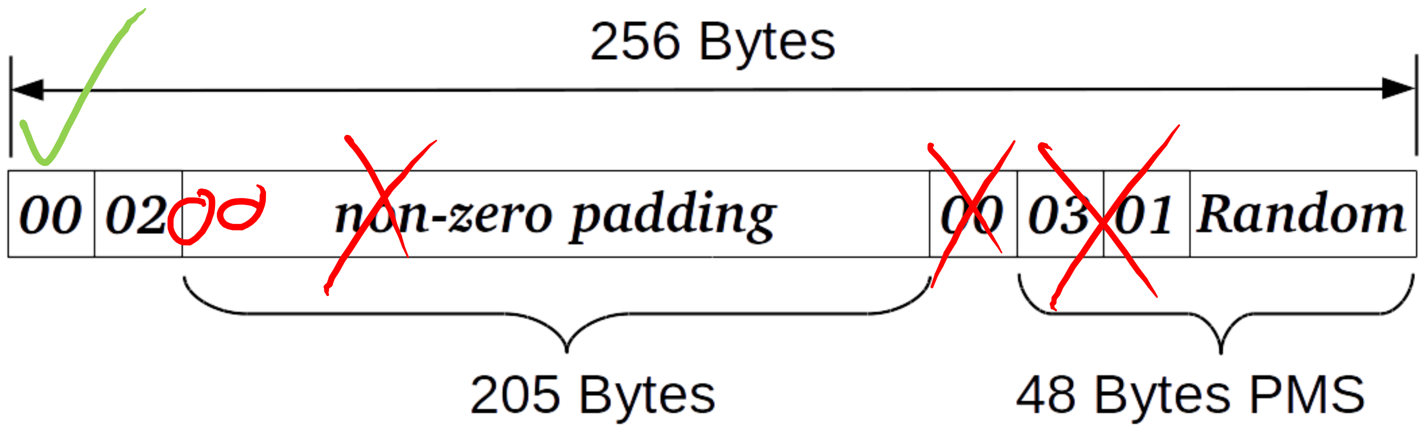


Bleichenbacher's attack enables adversary (in possession of an RSA ciphertext c_0) to recover the plaintext m_0

- Only prerequisite for this attack is the ability to access an oracle O
  1. that decrypts a ciphertext c
  2. and responds with 1 or 0, depending on whether
     a) the decrypted message m starts with $0x00\ 0x02$
     b) or not
- If the oracle answers with 1, the adversary knows that $2B \leq m \leq 3B - 1$ with $B = 2^{8(l-2)}$

"Strength" of O

- Bleichenbacher's attack requires ciphertexts that decrypt to plaintexts beginning with $0x00\ 0x02$
- But: PKCS#1 v1.5 performs several more checks besides the initial two bytes
- → Fewer checks results in stronger O



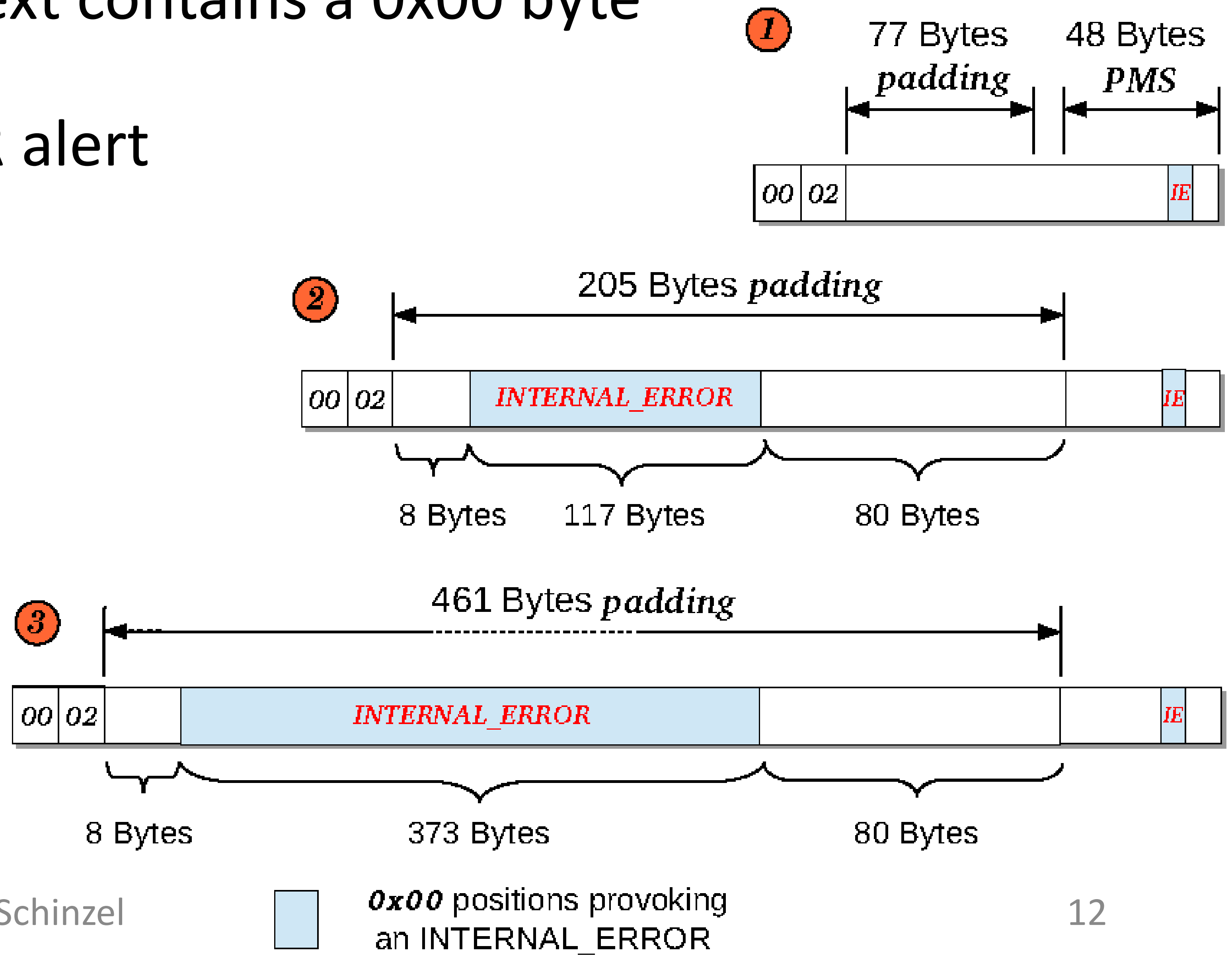
Countermeasures for Bleichenbacher's attack

- Idea:
  - "Let's stick to PKCS#1 v1.5 padding for compatibility reasons!"
  - "But: Make processing of valid records and invalid records indistinguishable"
- → Unify all error conditions and prevent attacker from creating a Bleichenbacher oracle

Revisiting SSL/TLS implementations

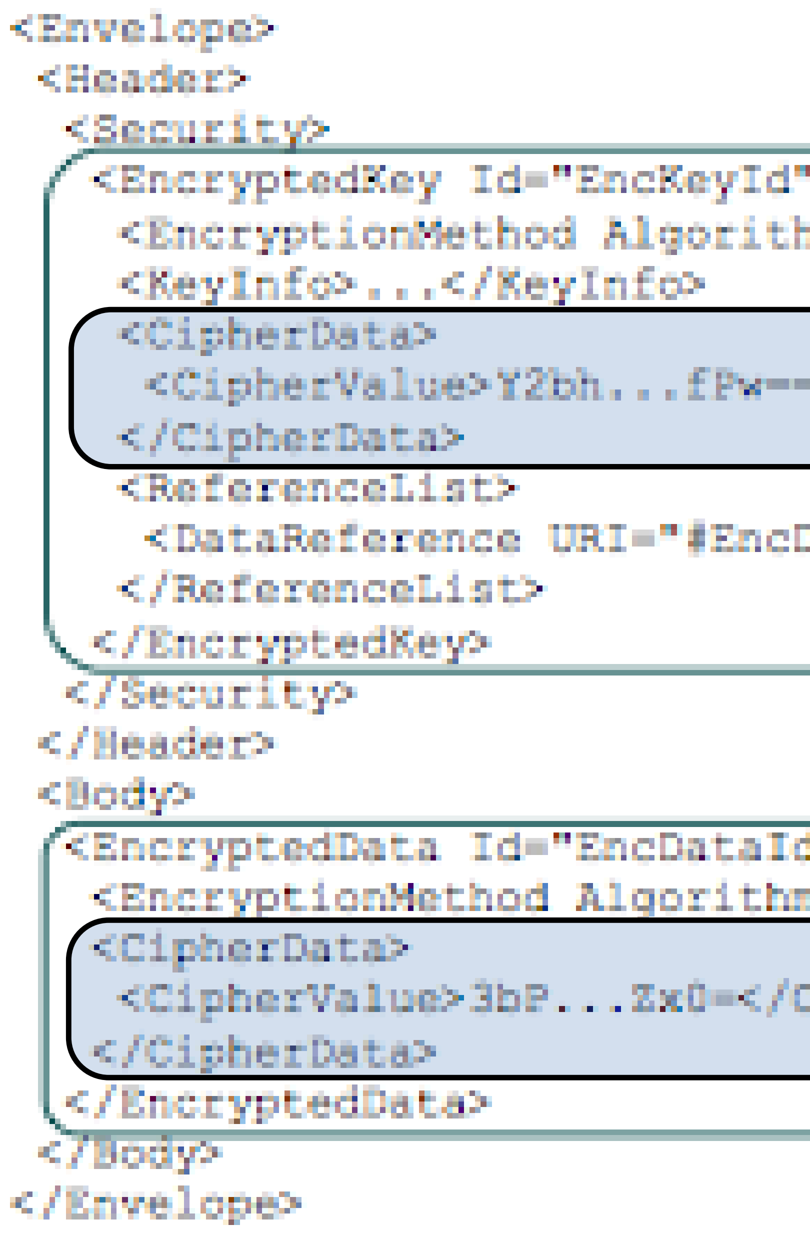
First channel:
Distinguishable error message in Java Secure Socket Extension (JSSE)

- IF: cleartext starts with 0x00 02 AND cleartext contains a 0x00 byte preceded with non-0x00 bytes
- THEN JSSE responds with INTERNAL_ERROR alert
- O Strength:
  - ~0,2% for 1024 bit keys
  - ~36% for 2048 bit keys
  - ~74% for 4096 bit keys
- Attack performance:
  - Hundreds of millions for 1024 bit keys
  - 176.797 requests for 2048 bit keys (12 hours)
  - 73.710 requests for 4096 bit keys (6 hours)



Related work: Bleichenbacher attack against XML Encryption

- Bleichenbacher’s original paper from 1998 exploited explicit TLS error handling, but he suggested that timing channels might be possible
- First timing-based Bleichenbacher attack was against XML Encryption in 2012

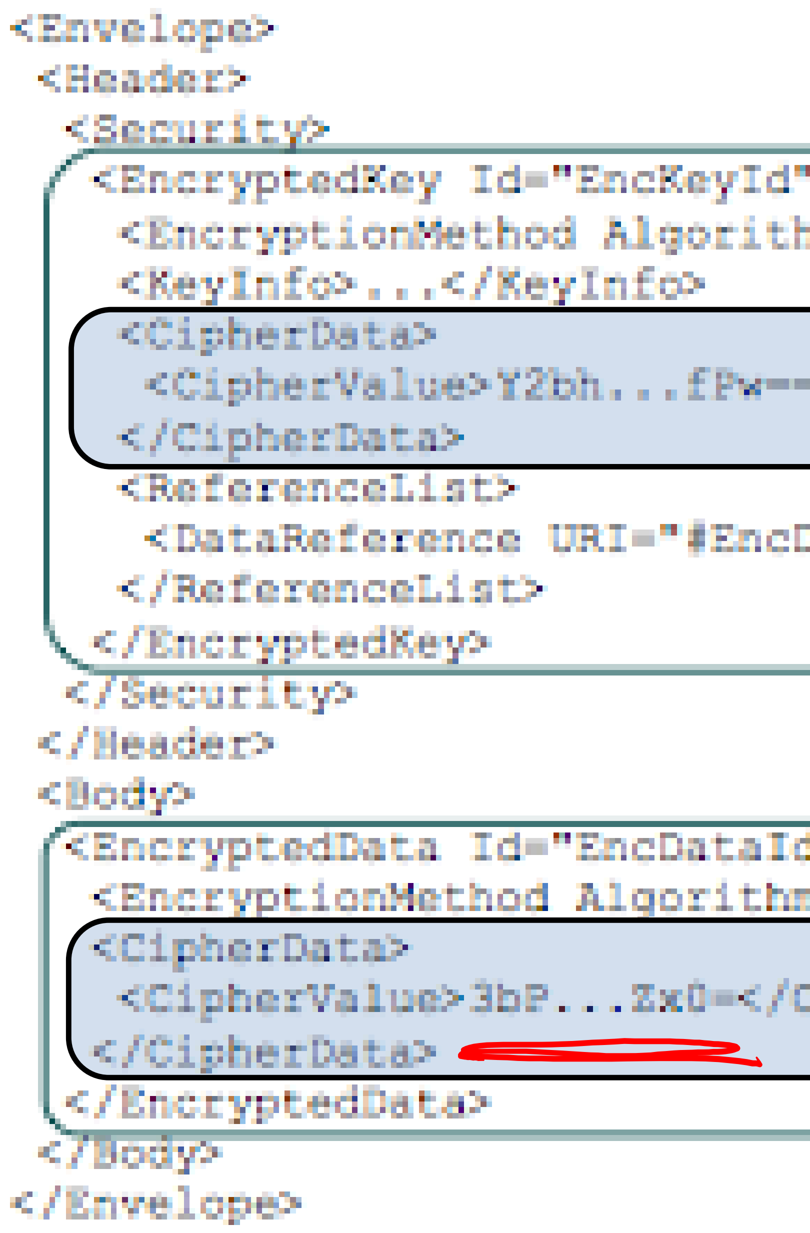


Tibor Jager, Sebastian Schinzel, Juraj Somorovsky
Bleichenbacher's Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption
17th European Symposium on Research in Computer Security (ESORICS 2012)
http://www.nds.rub.de/research/publications/breaking-xml-encryption-pkcs15/

Related work: Bleichenbacher attack against XML Encryption

- Decrypting XML Encryption messages

1. Decrypt session key $m = dec_{rsa}(c_{key})$
2. Return error if m does not comply with PKCS#1, else: ←
3. Decrypt c_{data} (results in XML subtree)
4. Copy subtree in XML doc
5. Parse XML doc
6. Return error if XML doc is invalid ←



→ Determine PKCS#1 compliance through response time

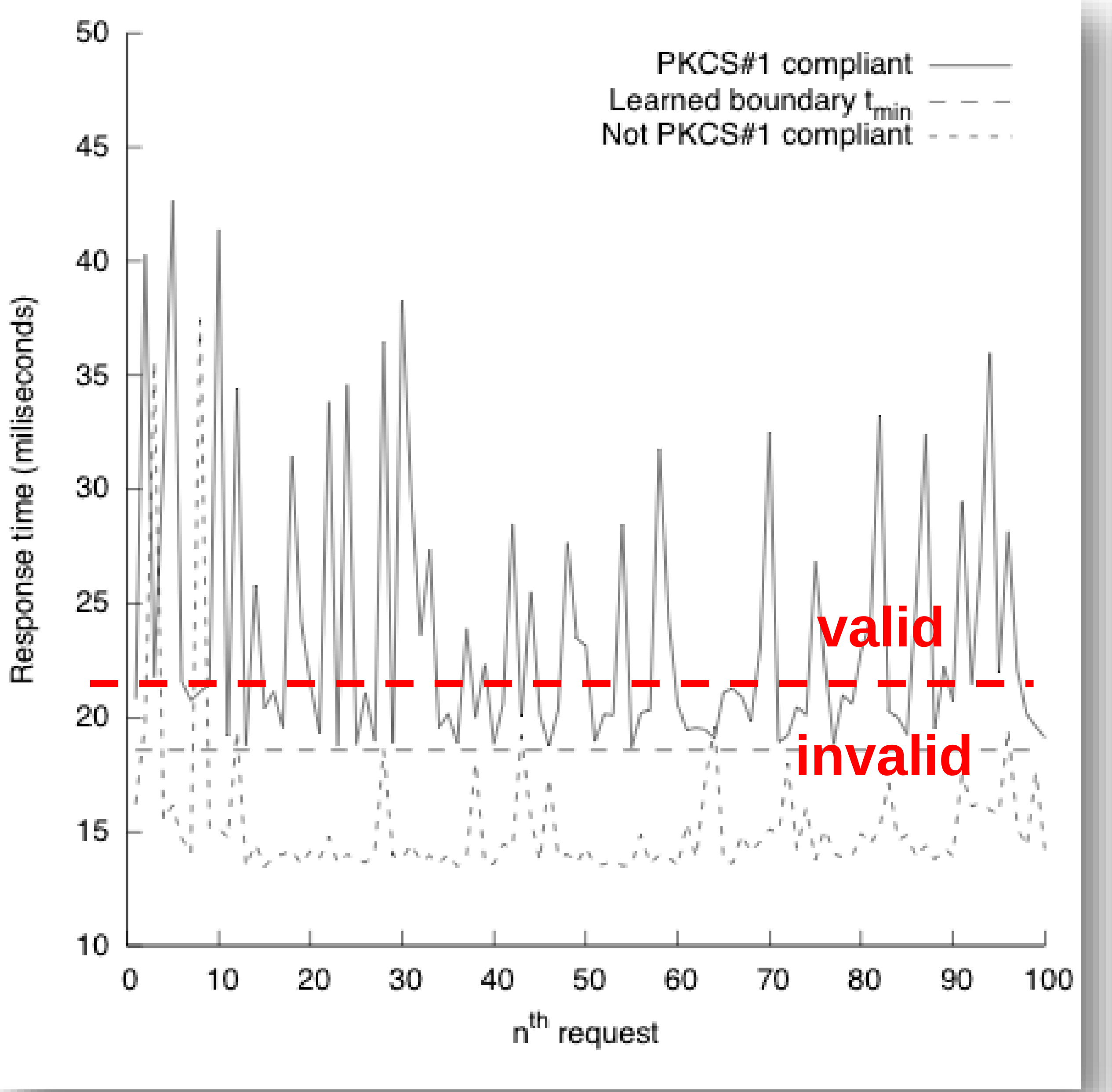
Related work: Bleichenbacher attack against XML Encryption

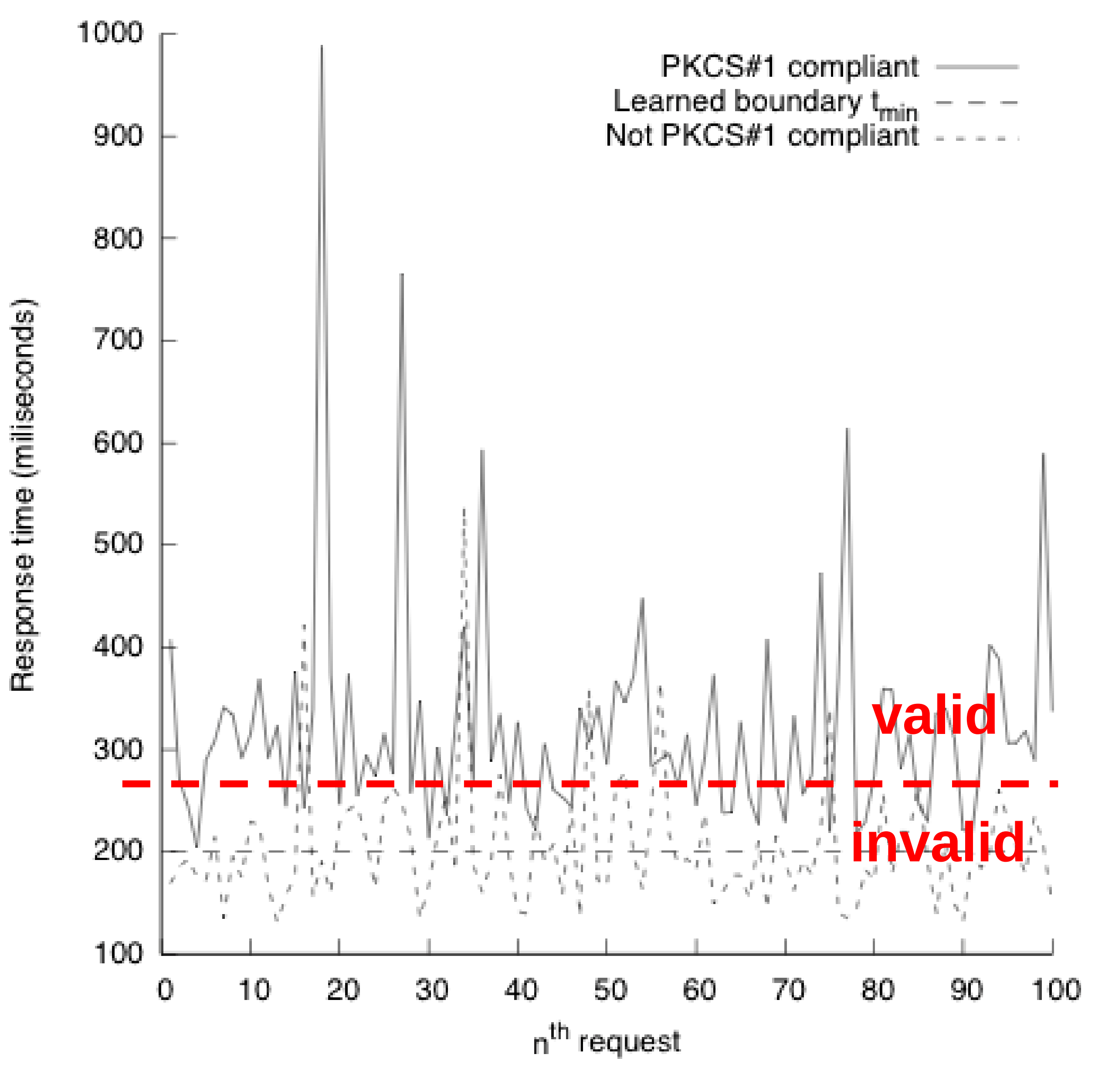
Results:
Bleichenbacher timing oracle

398,123 server requests

Localhost:
→ less than 200 minutes

Internet:
→ less than 1 week





Countermeasures for Bleichenbacher's attack

- Idea: Make processing of valid records and invalid records indistinguishable
- How does the current TLS version (1.2) deal with Bleichenbacher's attack? RFC5246:

```
1. Generate a string R of 48 random bytes
2. Decrypt the message to recover the plaintext M
3. If the PKCS#1 padding is not correct:
           pre_master_secret = R
       else If [...]
           [...]
       else:
           premaster secret = M
```

Countermeasures for Bleichenbacher's attack

- Generate random key PMS_R. In case of PKCS#1 v1.5-invalid c, proceed with PMS_R in protocol
- PMS_R is always generated even if c is PKCS#1 v1.5-compliant
- provokes error condition in later stage in protocol

```
generate a random PMS_R
decrypt the ciphertext: m := dec(c)
if ( (m ≠ 00||02||PS||00||k) OR (|k| ≠ 48)
      OR (k_1||k_2 ≠ maj||min) then
    proceed with PMS := PMS_R
else
    proceed with PMS := k
end if
```

Countermeasures for Bleichenbacher's attack

- What about TLS 1.0 and TLS 1.1?

The best way to avoid vulnerability to this attack is to treat incorrectly formatted messages in a manner indistinguishable from correctly formatted RSA blocks. Thus, **when it receives an incorrectly formatted RSA block, a server should generate a random 48-byte value and proceed using it** as the premaster secret. Thus, the server will act **identically** whether the received RSA block is correctly encoded or not.

Countermeasures for Bleichenbacher's attack

- TLS 1.0 and TLS 1.1 propose a slightly different schema:
- In case of PKCS#1 v1.5-invalid c generate random PMS_R and proceed in protocol
- PMS_R is only then generated if and only if c is not PKCS#1 v1.5-compliant

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

Let’s do some timing measurements!

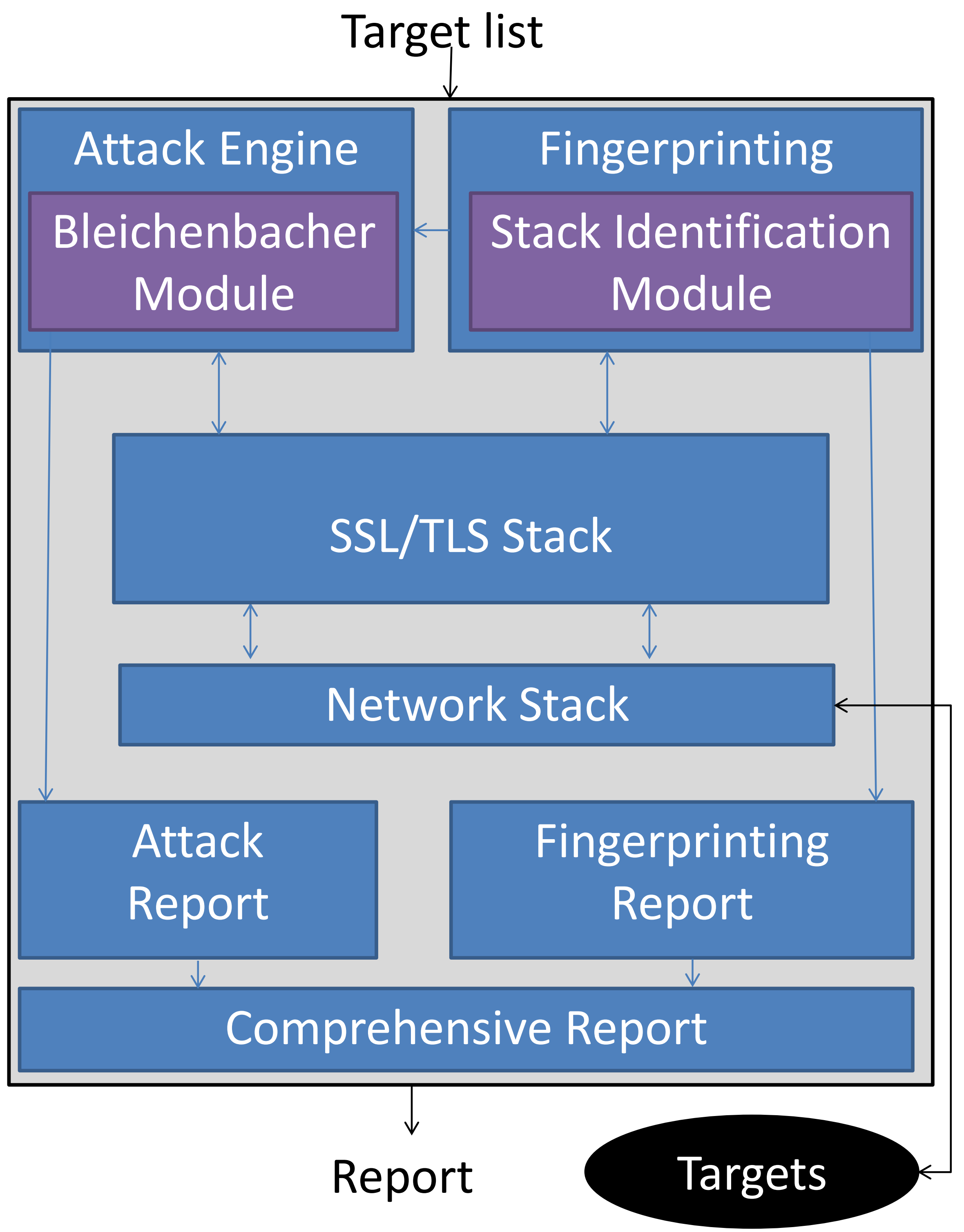
But, how can I perform timing attacks?
→ See my 28c3 talk “Time is on my side”

How can I (not) prevent timing leaks?
→ See my 29c3 talk “Time is not on your side”

Revisiting SSL/TLS implementations

T.I.M.E. TLS testing framework

- Credit to Chris Meyer
- Allows fine-grained construction of TLS test cases
- Very nice for fuzzing
- buuut: written in Java!



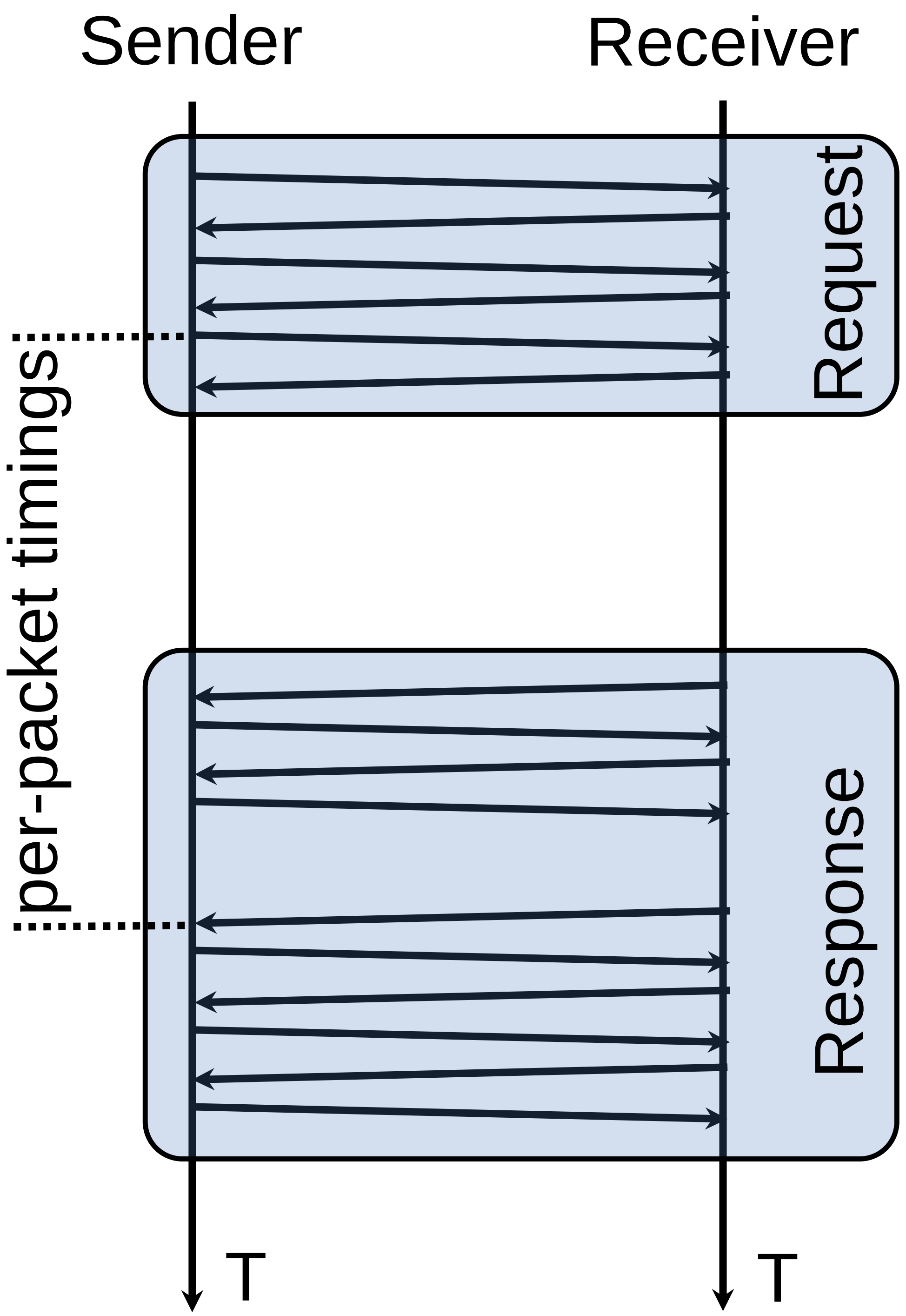
Timing measurement setup in a nutshell

- No memory-managed programming languages. Use C, Assembler, etc.
- Choose your part of the network wisely
  – no wireless; as near as possible to target; high quality routing hardware
- Disable power management
  – Intel SpeedStep (use “`cpufreq-utils`“ on Linux to fix frequency)
  – CPU C states (use “`idle=poll`” kernel boot parameter on Linux)
- Use old and cheap network interfaces (e.g. RTL 8139)
  – → No interrupt coalescing
- Stop all tasks and daemons on your local machine, no GUI
- Skip the first few hundred measurements (cache warm-up)

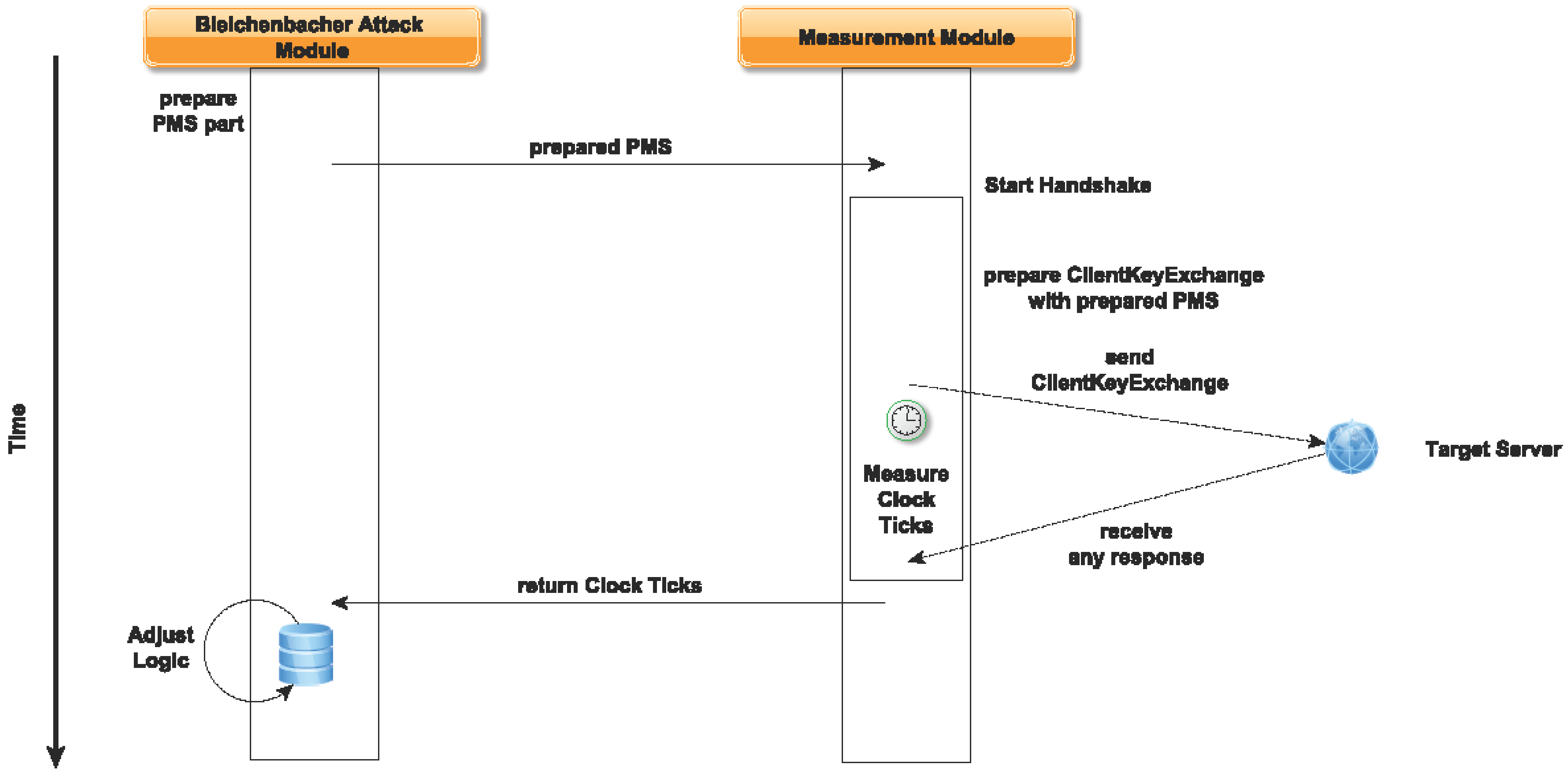
Timing measurement setup in a nutshell

Starting & end point
for measurements

1. send n-1 bytes of request
2. ⌛ start timer
3. send last byte of request
4. wait for receival of n^{th} byte of response
5. ⌛ stop timer



Measurement setup



Timing measurements with a patched version of the TLS implementation MatrixSSL

Patched MatrixSSL version that performs timing measurements (1/3)

- MatrixSSL's codebase is relatively clean
- No complex API wrappers
- Just `send()` and `recv()`

```
$ ./client base64(pms)
=== INITIAL CLIENT SESSION ===
We're sending info
Got state: 0
We were receiving info after 653680 ticks
Validated cert for:   Sample Matrix
                      RSA-1024 Certificate.
PMS is now encrypted
We're sending info
Got state: 0
We were receiving info after 3088811 ticks
FAIL: No HTTP Response
```

Patched MatrixSSL version that performs timing measurements (2/3)

- Sending data and setting the “start” timer

```
while ((len = matrixSslGetOutdata(ssl, &buf)) > 0) {
    transferred = send(fd, buf, len, 0);
    // Sebastian: Timestamp of the measurement start
    asm volatile(
        "cpuid \n"
        "rdtsc"
        : "=a"(minor),
          "=d"(mayor)
        : "a" (0)
        : "%ebx", "%ecx"
    );
    start = ((((ticks) mayor) << 32) | ((ticks) minor));
    // Sebastian: Start timestamp now in "start"
```

Patched MatrixSSL version that performs timing measurements (3/3)

- Receiving response and setting the “end” timer
- Roundtrip: `t=end-start`

```
if ((transferred = recv(fd, buf, len, 0)) < 0) {
    goto L_CLOSE_ERR;
}

// Sebastian: Timestamp of the measurement end
        asm volatile(
            "cpuid \n"
            "rdtsc"
            : "=a"(minor),
              "=d"(mayor)
            : "a" (0)
            : "%ebx", "%ecx"
        );
end = ((((ticks) mayor) << 32) | ((ticks) minor));
// Sebastian: End timestamp now in "end"
```

Second channel: Timing side channel in OpenSSL

- Let‘s look how OpenSSL treats Bleichenbacher‘s attack

Source code of OpenSSL 1.0.1i

s3_srvr.c:2216

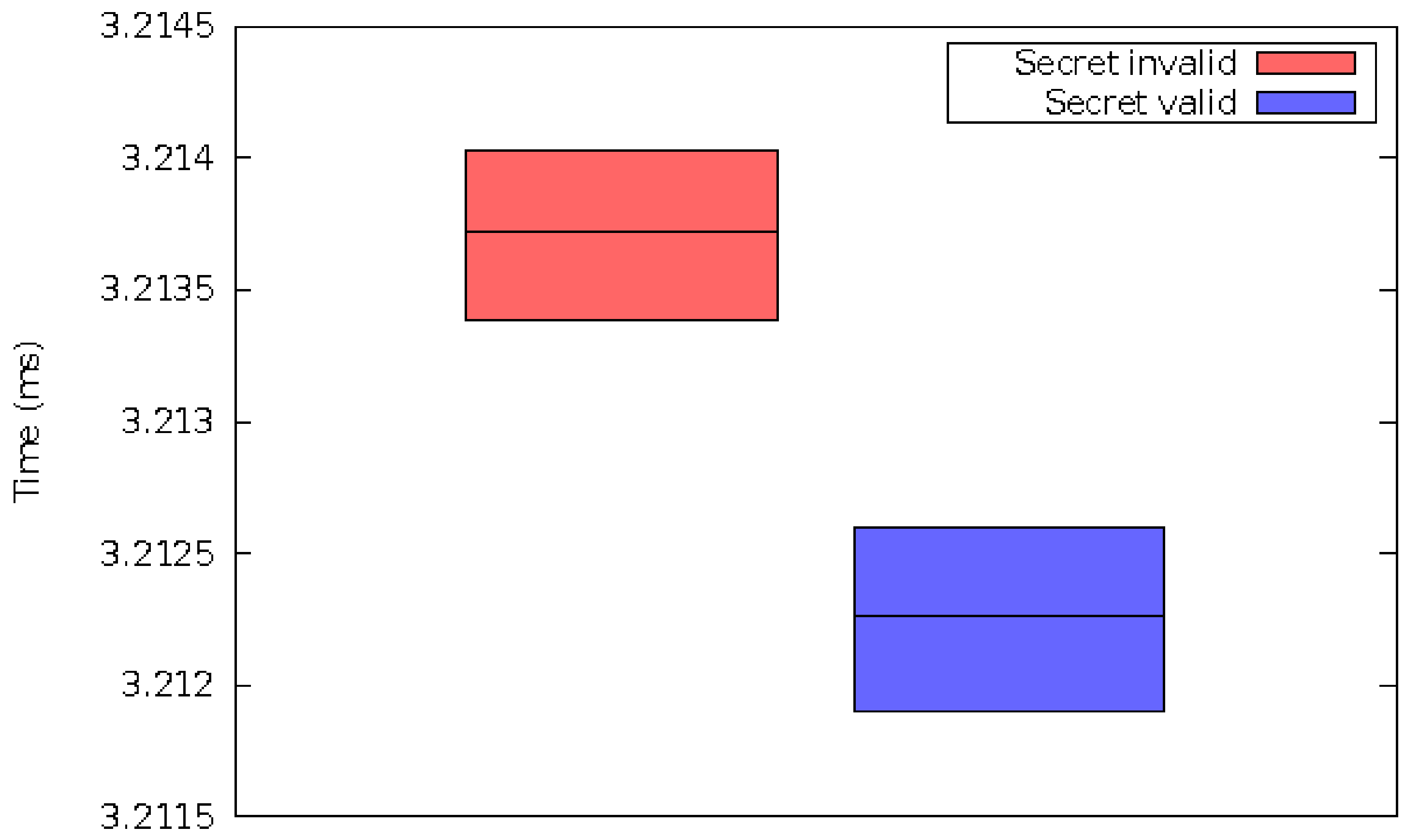
```
i=RSA_private_decrypt((int)n,p,p,rsa,RSA_PKCS1_PADDING);

al = -1;
```

s3_srvr.c:2251

```
if (al != -1)
    {
    /* Some decryption failure -- use random value instead as countermeasure
     * against Bleichenbacher's attack on PKCS #1 v1.5 RSA padding
     * (see RFC 2246, section 7.4.7.1). */
    ERR_clear_error();
    i = SSL_MAX_MASTER_KEY_LENGTH;
    p[0] = s->client_version >> 8;
    p[1] = s->client_version & 0xff;
    if (RAND_pseudo_bytes(p+2, i-2) <= 0) /* should be RAND_bytes, but we cannot work around a failure */
        goto err;
    }
    s->session->master_key_length = s->method->ssl3_enc->generate_master_secret(s, s->session->master_key
    OPENSSL_cleanse(p,i);
}
```

Second channel: Timing side channel in OpenSSL

- Generates random PMS if and only if cleartext was not PKCS#1-compliant
- ~1,5 microseconds delta
- O Strength: very weak: $2{,}7 * 10^{-8}$
- Attack performance (estim.): $5 * 10^{12}$ requests



Revisiting SSL/TLS implementations

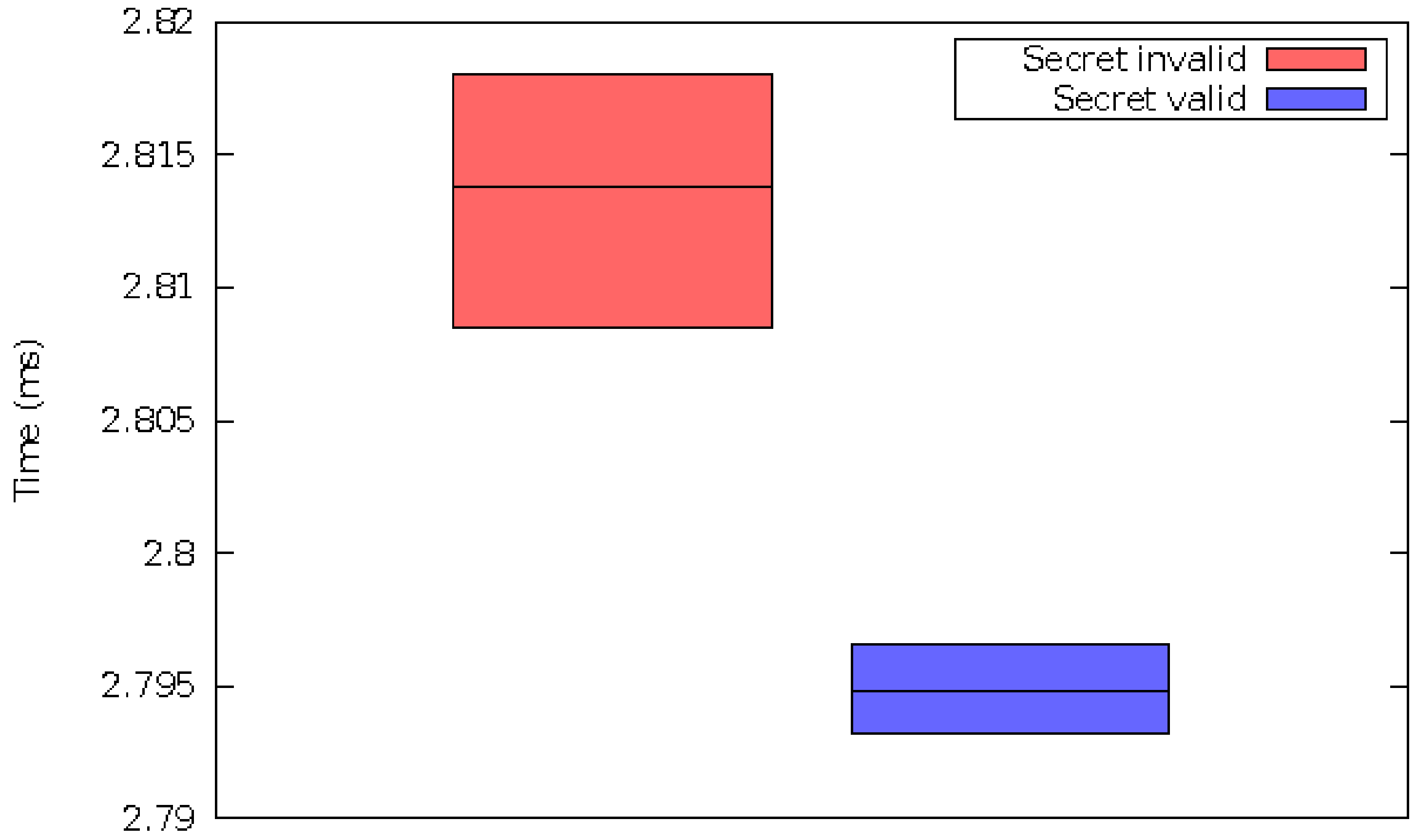
Third channel: Timing side channel in Java Secure Socket Extension (JSSE)

- Java's TLS impl.

```
SecretKey masterSecret;
try {
    KeyGenerator kg = JsseJce.getKeyGenerator("SunTlsMasterSecret");
    kg.init(spec);
    masterSecret = kg.generateKey();
} catch (GeneralSecurityException e) {
    // For RSA premaster secrets, do not signal a protocol error
    // due to the Bleichenbacher attack. See comments further down.
    if (!preMasterSecret.getAlgorithm().equals("TlsRsaPremasterSecret")) {
        throw new ProviderException(e);
    }
    if (debug != null && Debug.isOn("handshake")) {
        System.out.println("RSA master secret generation error:");
        e.printStackTrace(System.out);
    }
    preMasterSecret =
            RSAClientKeyExchange.generateDummySecret(protocolVersion);
    // recursive call with new premaster secret
    return calculateMasterSecret(preMasterSecret, null);
```

Third channel: Timing side channel in JSSE

- JSSE TLS implementation is textbook object-oriented, e.g. with exception handling
- O Strength:
  ~60% (very strong)
- Attack performance:
  18.600 requests (19,5 hours)

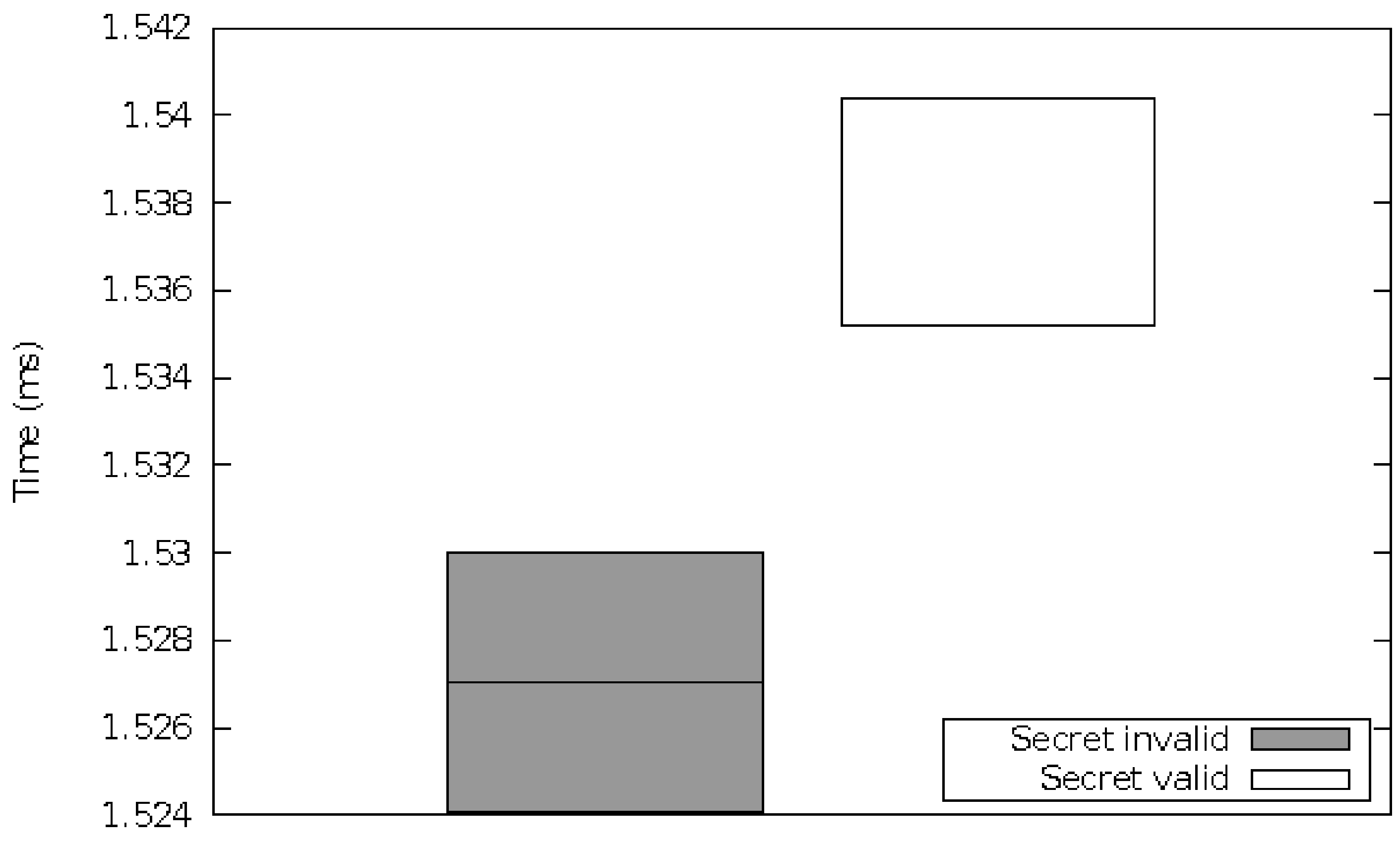


Fourth channel: Timing side channel in Cavium hardware TLS accelerators

- Processing of expensive crypto operations is performed on separate hardware
- Comes as PCI card
- Often used by big appliances that need to handle thousands of parallel TLS handshakes and connections

Fourth channel: Timing side channel in Cavium hardware TLS accelerators

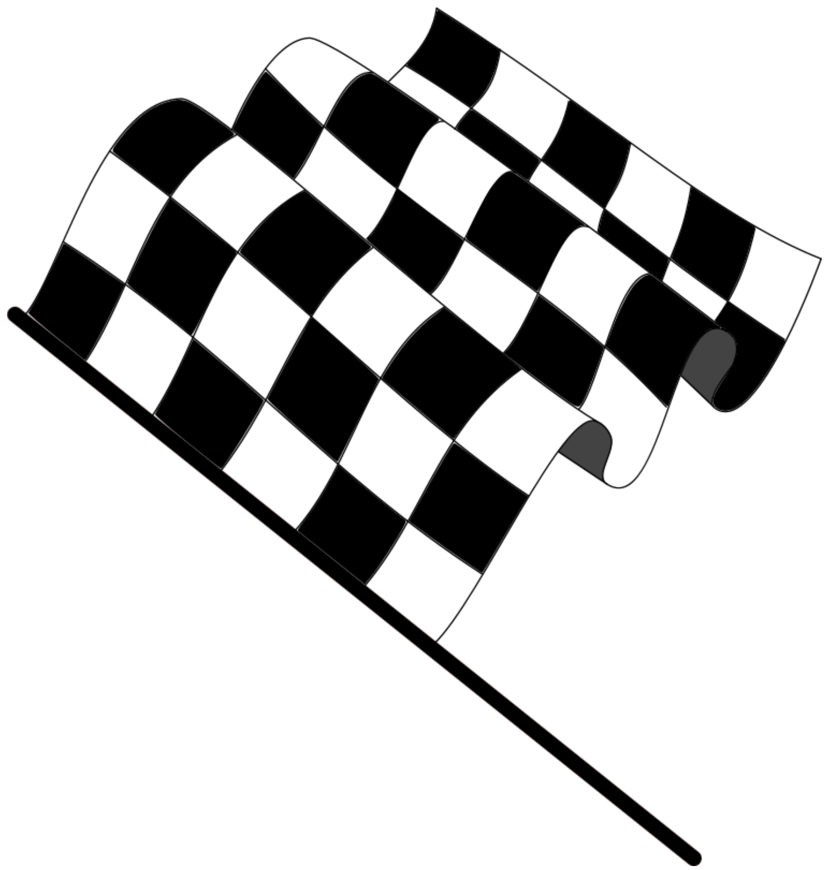
- e.g. used in F5 BIG-IP, IBM Datapower
- Doesn’t verify first byte, only second byte (0x?? 02)
- Needed extension to Bleichenbacher’s algorithm
- Attack performance: 4.000.000 queries (41 hours)



Summary:

- Timing attacks against single digit microsecond delays in TCP connections are practical in local networks
- Bad designs in cryptographic protocols may taunt you for decades to come
  - MAC-then-encrypt *
  - RSA + PKCS#1 v1.5
  - …
- Implementing TLS is a minefield

* http://tools.ietf.org/html/draft-gutmann-tls-encrypt-then-mac-04



See you around at 31c3!