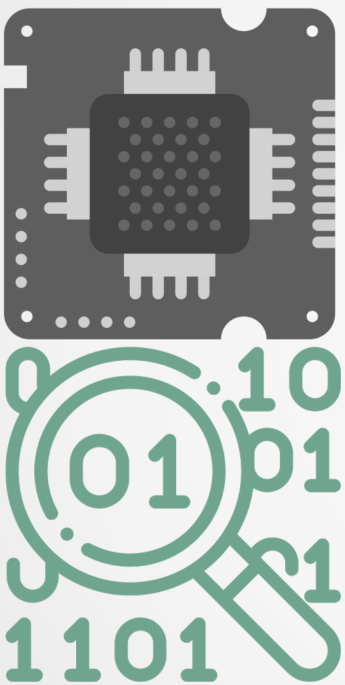
Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification

Sze Yiu Chau
Purdue University

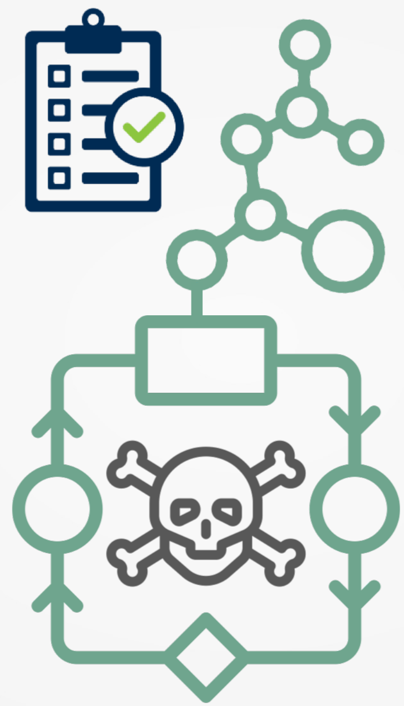
Joint work with Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, Ninghui Li

The University of Iowa
Purdue University

Software Testing



Low-level
Errors

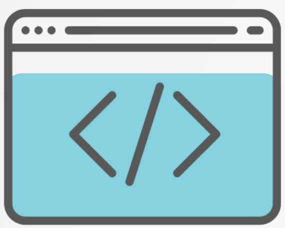


Logical Errors

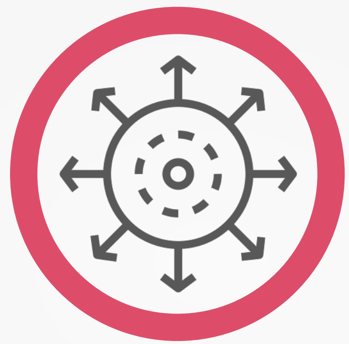
Attacks
e.g. libssh auth. bypass,
crypto. attacks

This work

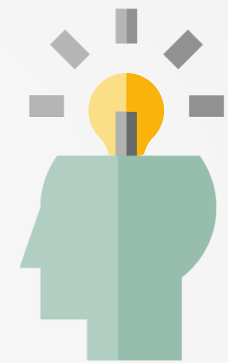
- Symbolic execution for analyzing semantic correctness



Code Coverage



Scalability Challenges

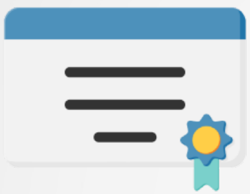


Domain Knowledge

- Case study: Implementing PKCS#1 v1.5 signature verification

Motivation

- PKCS#1 v1.5 RSA signature is widely used



...

- Previous work on X.509 testing neglect to analyze RSA signature verifications
  (Brubaker et al. 2014, Chen et al. 2015, Chau et al. 2017)

Motivation

- Implementations need to verify signatures robustly
- Logical errors may allow signature forgery
  - e.g. Bleichenbacher 2006, Kühn et al. 2008

Textbook RSA signature

- **Signing message m:**

m

H(m)

$H(m)^d \bmod n$ where d = private exponent
n = modulus

S

- **Given (S, m, e, n), verifying S is a valid signature of m**

S where e = public exponent

$S^e \bmod n \stackrel{?}{=} H(m)$

Beyond textbook RSA

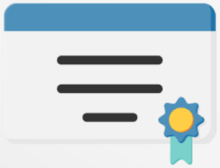
- Reality is more complex than that

1. Which H() to use?
    - SHA-1, SHA-2 family, SHA-3 family …
2. n is usually much longer than H(m)
    - $|n| \geq 2048$-bit
    - $|\text{SHA-1}| = 160$-bit, $|\text{SHA-256}| = 256$-bit

- Need padding and meta-data

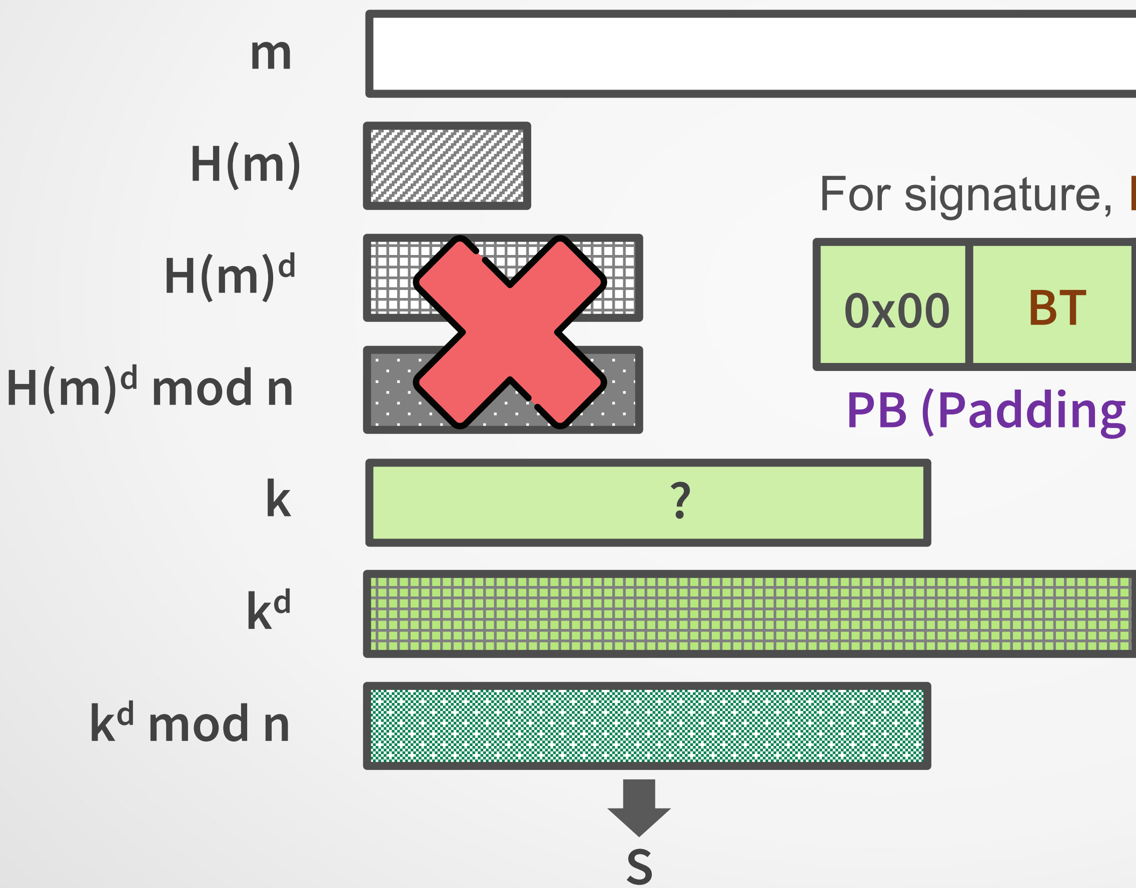
Beyond textbook RSA

- The PKCS#1 family of standards
  - Published by RSA (the company)
- Both encryption and signature schemes
  - version 2+ adapted schemes from Bellare et al.
- For signatures, version 1.5 most widely used
  - e.g. certificates of Google, Wikipedia



PKCS#1 v1.5 Signature Scheme

- **Signing:**



For signature, BT (Block Type) = 0x01

0x00	BT	PB	0x00	AS

PB (Padding Bytes) = 0xFF 0xFF … 0xFF

- At least 8-byte long
- Pad k to the size of n

AS is a DER-encoded ASN.1 structure, containing:

- Meta-data describing H()
- The actual H(m)

PKCS#1 v1.5 Signature Scheme

- DER encoded object is a tree of <T,L,V> triplets
- AS looks like this when encoded:

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w                        // ASN.1 SEQUENCE, length = w = 0x21
  /* [AlgorithmIdentifier] */
  30 x                      // ASN.1 SEQUENCE, length = x = 0x09
    06 u 2B 0E 03 02 1A     // ASN.1 OID, length = u = 0x05
    05 y           // ASN.1 NULL parameter, length = y = 0x00
  /* [Digest] */
  04 z               // ASN.1 OCTET STRING, length = z = 0x14
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

- altogether 35 bytes if H() = SHA-1()

PKCS#1 v1.5 Signature Scheme

- DER encoding is a tree of <T,L,V> triplets
- AS looks like this when encoded:

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w                        // ASN.1 SEQUENCE, length = w = 0x21
  /* [AlgorithmIdentifier] */
  30 x                      // ASN.1 SEQUENCE, length = x = 0x09
    06 u 2B 0E 03 02 1A     // ASN.1 OID, length = u = 0x05
    05 y          // ASN.1 NULL parameter, length = y = 0x00
  /* [Digest] */
  04 z                // ASN.1 OCTET STRING, length = z = 0x14
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

- altogether 35 bytes if H() = SHA-1()

PKCS#1 v1.5 Signature Scheme

- DER encoding is a tree of <T,L,V> triplets
- AS looks like this when encoded:

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w                      // ASN.1 SEQUENCE, length = w = 0x21
  /* [AlgorithmIdentifier] */
  30 x                    // ASN.1 SEQUENCE, length = x = 0x09
    06 u 2B 0E 03 02 1A     // ASN.1 OID, length = u = 0x05
    05 y            // ASN.1 NULL parameter, length = y = 0x00
  /* [Digest] */
  04 z                  // ASN.1 OCTET STRING, length = z = 0x14
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

- altogether 35 bytes if H() = SHA-1()

PKCS#1 v1.5 Signature Scheme

- DER encoding is a tree of <T,L,V> triplets
- AS looks like this when encoded:

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w                      // ASN.1 SEQUENCE, length = w = 0x21
  /* [AlgorithmIdentifier] */
  30 x                    // ASN.1 SEQUENCE, length = x = 0x09
    06 u 2B 0E 03 02 1A       // ASN.1 OID, length = u = 0x05
    05 y            // ASN.1 NULL parameter, length = y = 0x00
  /* [Digest] */
  04 z                // ASN.1 OCTET STRING, length = z = 0x14
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

- altogether 35 bytes if H() = SHA-1()

PKCS#1 v1.5 Signature Scheme

- DER encoding is a tree of <T,L,V> triplets
- AS looks like this when encoded:

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w                      // ASN.1 SEQUENCE, length = w = 0x21
  /* [AlgorithmIdentifier] */
  30 x                    // ASN.1 SEQUENCE, length = x = 0x09
    06 u 2B 0E 03 02 1A       // ASN.1 OID, length = u = 0x05
    05 y            // ASN.1 NULL parameter, length = y = 0x00
  /* [Digest] */
  04 z                // ASN.1 OCTET STRING, length = z = 0x14
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

- altogether 35 bytes if H() = SHA-1()

PKCS#1 v1.5 Signature Scheme

- DER encoding is a tree of <T,L,V> triplets
- AS looks like this when encoded:

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w                     // ASN.1 SEQUENCE, length = w = 0x21
  /* [AlgorithmIdentifier] */
  30 x                   // ASN.1 SEQUENCE, length = x = 0x09
    06 u 2B 0E 03 02 1A      // ASN.1 OID, length = u = 0x05
    05 y           // ASN.1 NULL parameter, length = y = 0x00
  /* [Digest] */
  04 z                 // ASN.1 OCTET STRING, length = z = 0x14
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

- altogether 35 bytes if H() = SHA-1()

PKCS#1 v1.5 Signature Scheme

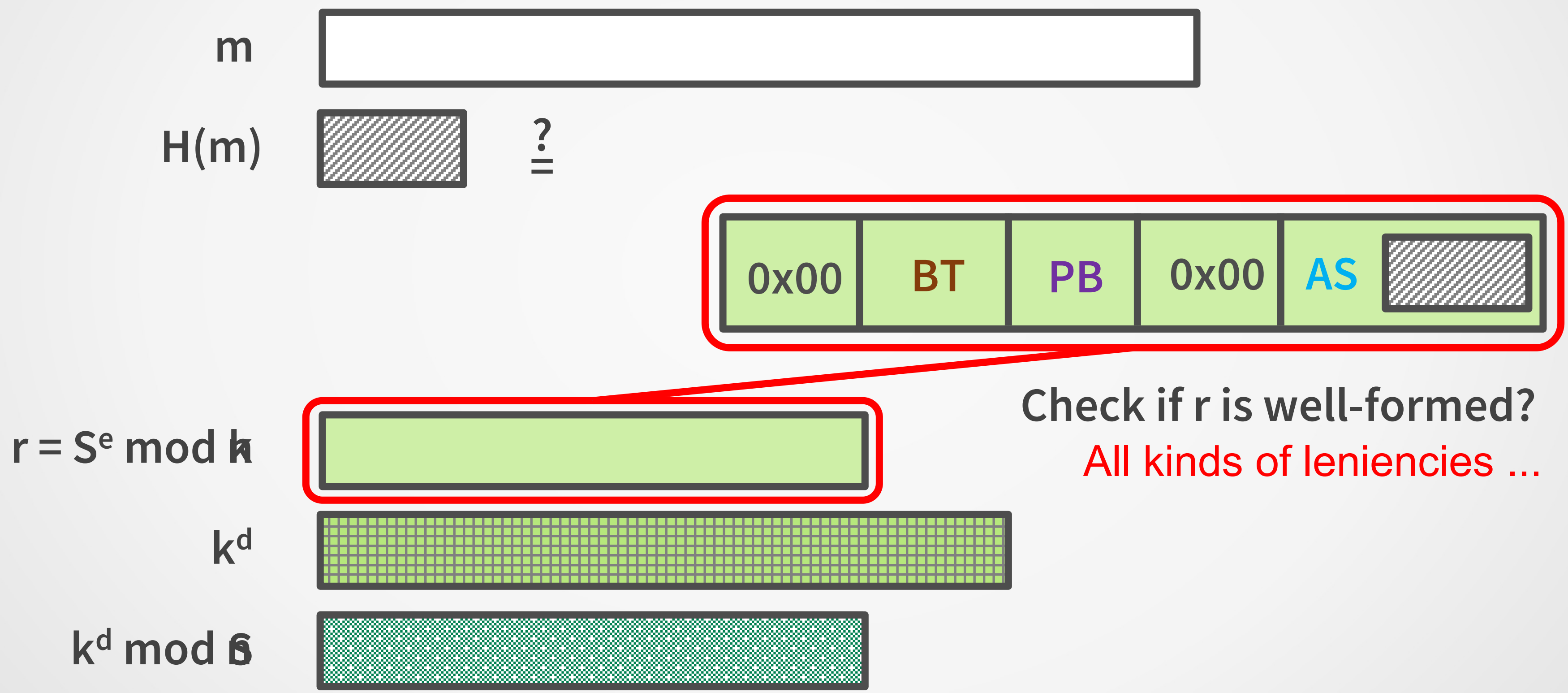
- DER encoding is a tree of <T,L,V> triplets
- AS looks like this when encoded:

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w                    // ASN.1 SEQUENCE, length = w = 0x21
  /* [AlgorithmIdentifier] */
  30 x                  // ASN.1 SEQUENCE, length = x = 0x09
    06 u 2B 0E 03 02 1A    // ASN.1 OID, length = u = 0x05
    05 y          // ASN.1 NULL parameter, length = y = 0x00
  /* [Digest] */
  04 z                // ASN.1 OCTET STRING, length = z = 0x14
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

- altogether 35 bytes if H() = SHA-1()

PKCS#1 v1.5 Signature Scheme

- Given (S, m, e, n), verifier computes H(m) and $r = S^e \bmod n$



Bleichenbacher's low exponent attack

- Yet another crypto attack attributed to D. Bleichenbacher
- CRYPTO 2006 rump session
- Some implementations accept malformed r'

r'	0x00	BT	PB	0x00	AS		Garbage

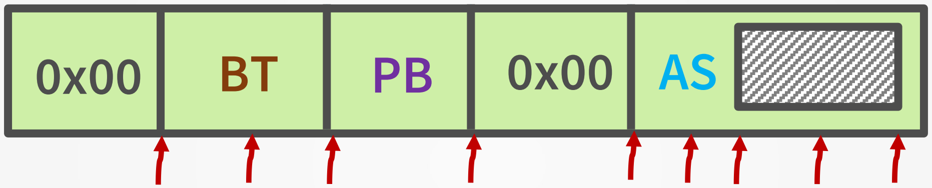
- Existential forgery possible when e is small
  - Generate signatures for arbitrary m without d

Bleichenbacher's low exponent attack

- A contributing factor to the push for bigger e (e.g. 65537)
- Smaller e more efficient for signature verifier
  - e = 3 prescribed in some protocols
    - (e.g. DNSSEC [RFC3110, Sect. 4])

Why was the attack possible?

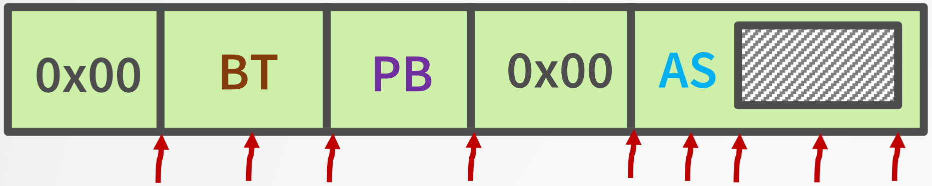
- Problem: accept malformed input w/ GARBAGE unchecked
  - Can be in many different locations, not only at the end



- Longer modulus makes forgery easier
  - More GARBAGE bits to use
  - Can handle longer hashes / slightly larger e

To find these attacks

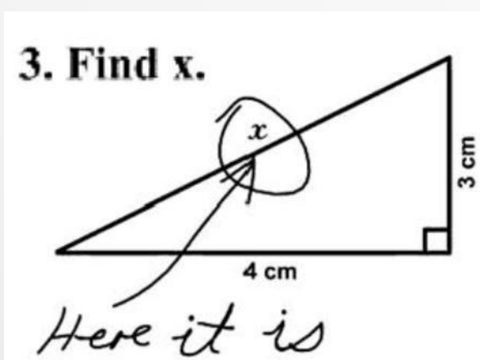
- Want to see how input bytes are being checked

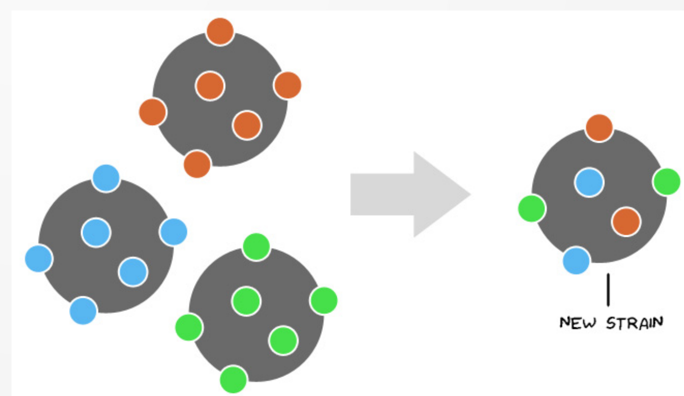


- If enough unchecked GARBAGE then

Automatically generate concolic test cases

- Observation: size of components exhibit linear relations
  - e.g. $\sum$ length(components)) = |n|;   ASN.1 DER
- Programmatically capture such linear constraints
- Ask Symbolic Execution to find satisfiable solutions

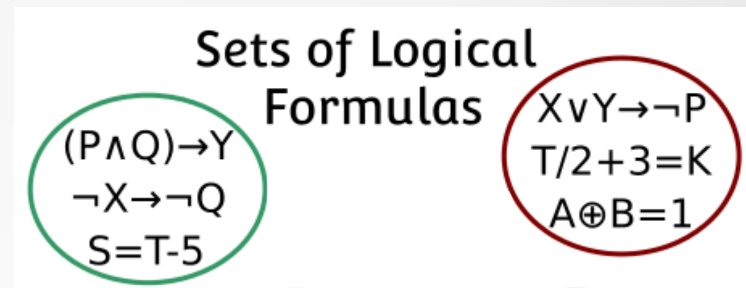




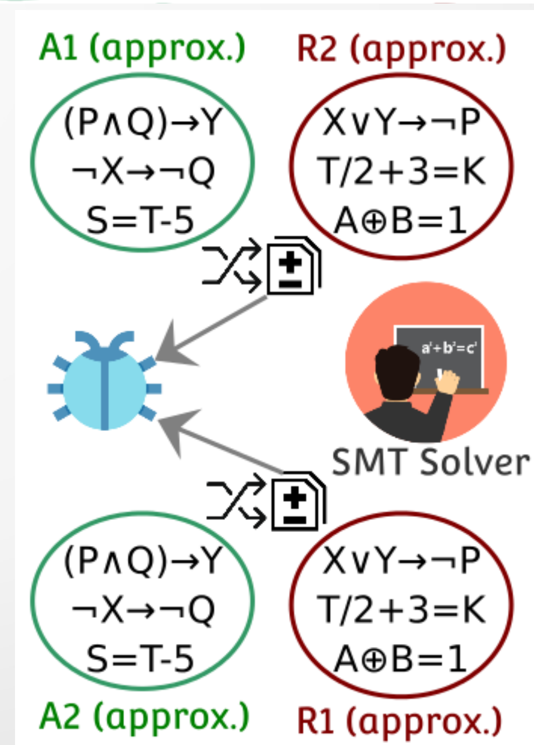
- Based on that, automatically pack symbolic/concrete components into test buffers

Testing with Symbolic Execution

- Symbolic Execution with concolic test cases

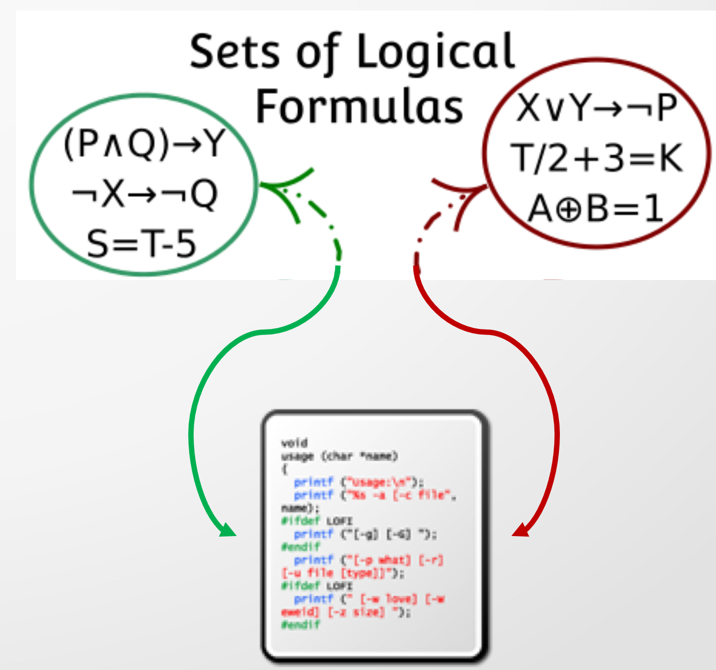


- Very useful abstraction
  - What and how things are being checked in code?
- Formulas can help
  cross-validate implementations



Finding root causes

- Locate the piece of code that imposes wrong constraints
- Can we go from formula abstraction back to code?
- Constraint Provenance Tracking
  - Keep a mapping of <clause, source-level origin>
  - Tiny space & time overhead



Implementations Tested

Discussion of signature forgery assumes e = 3 and SHA-1, attacks also applicable to newer hash algorithms

Name - Version	*Overly lenient*	*Practical exploit under small e*
axTLS - 2.1.3	YES	YES
BearSSL - 0.4	No	-
BoringSSL – 3112	No	-
Dropbear SSH – 2017.75	No	-
GnuTLS – 3.5.12	No	-
LibreSSL – 2.5.4	No	-
libtomcrypt – 1.16	YES	YES
MatrixSSL – 3.9.1 (Certificate)	YES	No
MatrixSSL – 3.9.1 (CRL)	YES	No
mbedTLS – 2.4.2	YES	No
OpenSSH – 7.7	No	-
OpenSSL – 1.0.2l	No	-
Openswan – 2.6.50 *	YES	YES
PuTTY – 0.7	No	-
strongSwan – 5.6.3 *	YES	YES
wolfSSL – 3.11.0	No	-

* configured to use their own internal implementations of PKCS#1 v1.5

Leniency in Openswan 2.6.50

- Ignoring padding bytes (CVE-2018-15836)
- Simple oversight, severe implications
  - Exploitable for signature forgery

```
/* check signature contents */
/* verify padding (not including
   any DER digest info!    */
padlen = sig_len - 3 - hash_len;
... ...
/* skip padding */
if(s[0] != 0x00 || s[1] != 0x01
   || s[padlen+2] != 0x00)
{ return "3""SIG padding does not
s += padlen + 3;
```

- Use this r'

```
/** all numbers below are hexadecimals **/
00 01 GARBAGE 00 30 21 … … 04 16 SHA-1(m')
```

- Want: $(a+b)^3 = a^3 + 3a^2b + 3b^2a + b^3$, s.t.
  - MSBs of a^3 give what is before GARBAGE
  - LSBs of b^3 give what is after GARBAGE
  - LSBs of $a^3 + 3a^2b + 3b^2a$ + MSBs of b^3 stay in GARBAGE
  - fake signature $S' = (a+b)$

New unit test in Openswan

xelerance / **Openswan**

<> Code | Issues 95 | Pull requests 0 | Projects 0 | Wiki | Insights

wo#7449 . test case for Bleichenbacher-style signature forgery

```
Special thanks to Sze Yiu Chau of Purdue University (schau@purdue.edu)
who reported the issue, and made major contributions towards defining
this test case.
```

master (#330) **v2.6.51.2** … v2.6.50.1

bartman committed on Aug 20 — 1 parent 9eaa6c2

Showing **6 changed files** with **218 additions** and **0 deletions**.

1 tests/unit/libopenswan/Makefile

```
@@ -23,6 +23,7 @@ clean check:
23   23           @${MAKE} -C lo04-verifypubkeys   $@
24   24           @${MAKE} -C lo05-datatot         $@
25   25           @${MAKE} -C lo06-verifybadsigs $@
     26  +        @${MAKE} -C lo07-bleichenbacher-attack $@
26   27
```

Leniency in strongSwan 5.6.3

1. Not checking **AlgorithmParameter** (CVE-2018-16152)
    - classical flaw found in others like GnuTLS, Firefox years ago
    - Exploitable for signature forgery
        - hide GARBAGE in AlgorithmParameter
        - follow the Openswan attack algorithm
            - adjust what a^3 and b^3 represent, fake signature $S' = (a+b)$

Leniency in strongSwan 5.6.3

2. Accept trailing bytes after **Algorithm OID** (CVE-2018-16151)
    - interestingly, Algorithm OID is not matched exactly
    - a variant of longest prefix match

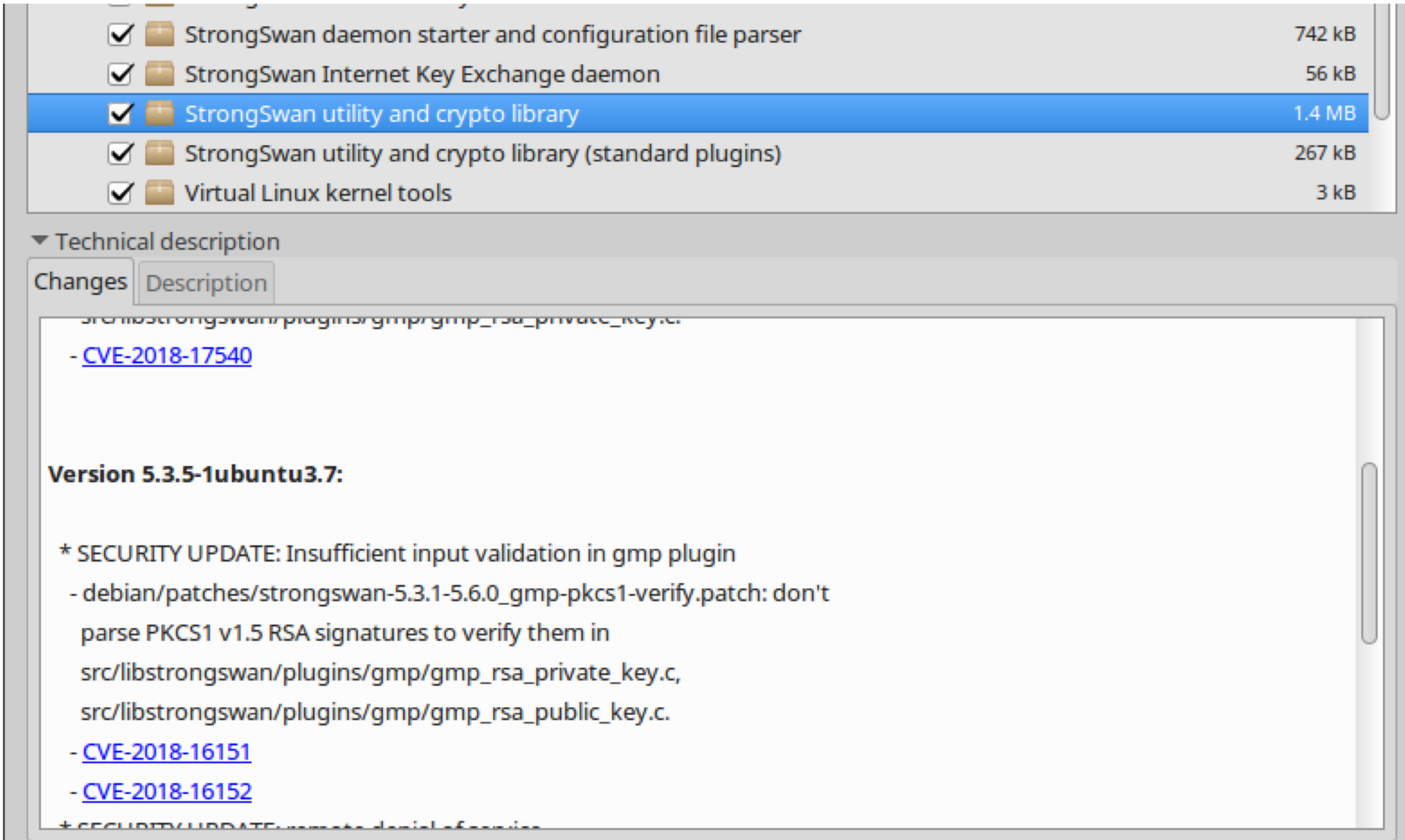
```
/* [AlgorithmIdentifier] */
30 09
   06 05 2B 0E 03 02 1A
   05 00
```

```
/* [AlgorithmIdentifier] */
30 0C
   06 08 2B 0E 03 02 1A AB CD EF
   05 00
```

both would be recognized as OID of SHA-1

- knowing this, one can hide **GARBAGE** there
- follow the Openswan attack algorithm
    - adjust what a^3 and b^3 represent, **fake signature** $S' = (a+b)$

strongSwan Security Update



Some key generation programs still forces e = 3

e.g., ipsec_rsasigkey on Ubuntu

NAME

ipsec_rsasigkey - generate RSA signature key

SYNOPSIS

ipsec rsasigkey [--verbose] [--seeddev device] [--seed numbits] [--nssdir nssdir]
[--password nsspassword] [--hostname hostname] [nbits]

DESCRIPTION

rsasigkey generates an RSA public/private key pair, suitable for digital signatures, of (exactly) nbits bits (that is, two primes each of exactly nbits/2 bits, and related numbers) and emits it on standard output as ASCII (mostly hex) data. nbits must be a multiple of 16.

The public exponent is forced to the value 3, which has important speed advantages for signature checking. Beware that the resulting keys have known weaknesses as encryption keys **and should not be used for that purpose.**

Leniency in axTLS 2.1.3

1. Accepting trailing **GARBAGE** (CVE-2018-16150)
   - original Bleichenbacher '06 forgery also works

Leniency in axTLS 2.1.3

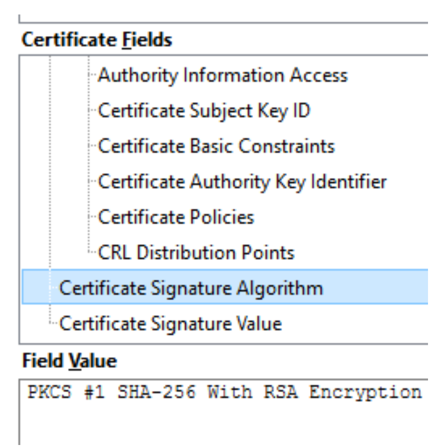
2. Ignoring AS.AlgorithmIdentifier (CVE-2018-16253)

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 21
  /* [AlgorithmIdentifier] */
  30 09
    06 05 2B 0E 03 02 1A
    05 00
  /* [Digest] */
  04 14
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

this whole chunk is skipped ...

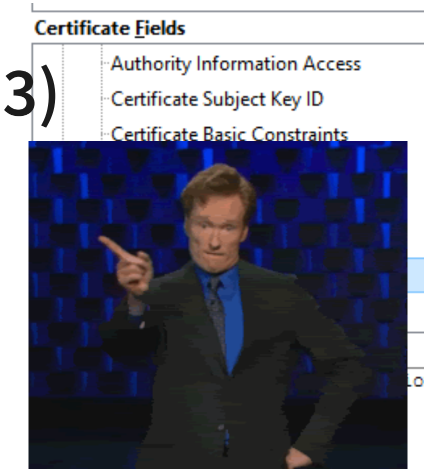
```
if (asn1_next_obj(asn1_sig, &offset,
    ASN1_SEQUENCE) < 0 ||
    asn1_skip_obj(asn1_sig, &offset,
    ASN1_SEQUENCE))     goto end_get_sig;

if (asn1_sig[offset++] != ASN1_OCTET_STRING)
    goto end_get_sig;
*len = get_asn1_length(asn1_sig, &offset);
ptr = &asn1_sig[offset];            /* all ok */

end_get_sig:
    return ptr;
```

- Probably because certificates have an explicit signature algorithm field, which gives H()



Leniency in axTLS 2.1.3

2. Ignoring AS.AlgorithmIdentifier (CVE-2018-16253)



- Just because H() is known from outside
- Doesn't mean it can be skipped

- Use this r'

```
/** all numbers below are hexadecimals **/
00 01 FF FF FF FF FF FF FF FF 00
30 5D 30 5B GARBAGE 04 16 SHA-1(m')
```

  - hide **GARBAGE** in **AlgorithmIdentifier**
  - follow the Openswan attack algorithm
    - adjust what a^3 and b^3 represent, **fake signature** S' = (a+b)

Leniency in axTLS 2.1.3

3. **Trusting the declared ASN.1 DER lengths w/o sanity checks [CVE-2018-16149]**

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w
  /* [AlgorithmIdentifier] */
  30 x
    06 u 2B 0E 03 02 1A
    05 y
  /* [Digest] */
  04 z
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

put absurdly large values to trick verifier into reading from illegal addresses

- **DoS PoC: making z exceptionally large crashed the verifier**

patching axTLS (ESP8266 port)

igrr / **axtls-8266**

Code | Issues 8 | Pull requests 1 | Projects 0 | Insights

Apply CVE fixes for X509 parsing

```
Apply patches developed by Sze Yiu which correct a vulnerability in
X509 parsing.  See CVE-2018-16150 and CVE-2018-16149 for more info.
```

master (#60)

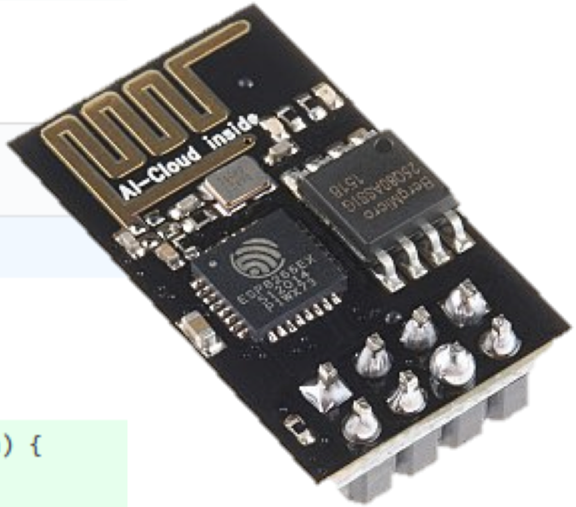
earlephilhower authored and **igrr** committed on Oct 22 — `1 parent e634adf`

Showing **2 changed files** with **76 additions** and **38 deletions**.

`12 ssl/os_port.h`

```
@@ -142,6 +142,18 @@ static inline int strlen_P(const char *str) {
142 142         while (pgm_read_byte(str++)) cnt++;
143 143         return cnt;
144 144     }
    145 + static inline int memcmp_P(const void *a1, const void *b1, size_t len) {
    146 +     const uint8_t* a = (const uint8_t*)(a1);
    147 +     uint8_t* b = (uint8_t*)(b1);
```

ESP8266



Other leniencies

- Lax checks on ASN.1 DER lengths in MatrixSSL(CRL)

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w
  /* [AlgorithmIdentifier] */
  30 x
    06 u 2B 0E 03 02 1A
    05 y
  /* [Digest] */
  04 z
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

many possible values will be accepted

- Some bits in the middle of AS can take any values
- Doesn’t seem to be numerous enough for practical attacks
- Variants of this leniency also found in *mbedTLS*, *libtomcrypt*, *MatrixSSL (Certificate)*

Leniency in MatrixSSL 3.9.1

MatrixSSL 4.x changelog

Changes between 4.0.0 and 4.0.1 [November 2018]

This version improves the security of RSA PKCS #1.5 signature verification and adds better support for run-time security configuration.

- Crypto:
  - Changed from a parsing-based to a comparison-based approach in DigestInfo validation when verifying RSA PKCS #1.5 signatures. There are no known practical attacks against the old code, but the comparison-based approach is theoretically more sound. Thanks to Sze Yiu Chau from Purdue University for pointing this out.
  - (MatrixSSL FIPS Edition only:) Fix DH key exchange when using DH parameter files containing optional privateValueLength argument.
  - psX509AuthenticateCert now uses the common psVerifySig API for signature verification. Previously, CRLs and certificates used different code paths for signature verification.

Conclusion

- RSA signature verification should be robust regardless of the choice of e
  - Flawed verification can break authentication in different scenarios

- To analyze this, we extend symbolic execution with
  - Automatic generation of concolic test cases
  - Constraint Provenance Tracking
- Found new variants of Bleichenbacher '06 attacks after more than a decade, 6 new CVEs
  - And many other unwarranted leniencies

Q&A

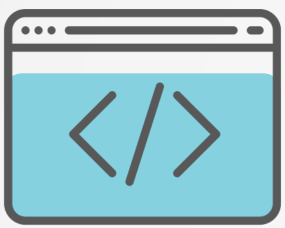
Thank You

Sze Yiu Chau
Computer Science, Purdue University
Email: schau@purdue.edu
Web: https://www.cs.purdue.edu/homes/schau/

Additional Slides

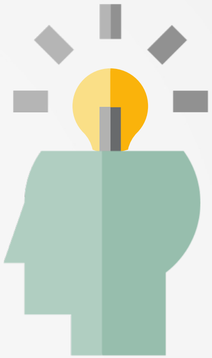
This work

- Symbolic execution for analyzing semantic correctness



Code Coverage

Scalability Challenges



Domain Knowledge

- Case study: Implementing PKCS#1 v1.5 signature verification
  - Flaws in TLS and crypto libraries, and IPSec software
  - Exploitable for signature forgery and DoS attacks

Bleichenbacher's low exponent attack

- r' = 0x00 || BT || PB || 0x00 || AS || GARBAGE
- Assuming e = 3
- Construct t = 00 01 FF FF FF FF FF FF FF FF 00 || AS || FF .. FF
  - if $(\text{floor}(\sqrt[3]{t}))^3$ match r' before GARBAGE then
    - fake signature S' = $\text{floor}(\sqrt[3]{t})$
  - if necessarily, tweak m to get a different AS, try again

Bleichenbacher's low exponent attack

- r' = 0x00 || BT || PB || 0x00 || AS || GARBAGE
  - = 00 01 FF FF FF FF FF FF FF FF 00 || AS || GARBAGE
  - assuming e = 3, |n| = 1024 bits, r' < 2^{1009}
    - want to find S' s.t. S'^3 = r'
- construct t = 00 01 FF FF FF FF FF FF FF FF 00 || AS || FF .. FF
  - if $(\text{floor}(\sqrt[3]{t}))^3$ match r' before GARBAGE then we're done
    - fake signature S' = $\text{floor}(\sqrt[3]{t})$
  - if necessarily, tweak m to get a different AS, try again

Bleichenbacher's low exponent attack

- r' = 00 01 FF FF FF FF FF FF FF FF 00 || AS || GARBAGE
  - assuming $e = 3$, $|n| = 1024$ bits, $r' < 2^{1009}$
    - want to find S' s.t. $S'^3 = r'$
- distance between two consecutive perfect cubes:
    - $b^3 - (b-1)^3 = 3b^2 - 3b + 1 < 3 \cdot 2^{673} - 3 \cdot 2^{337} + 1$
      $< 2^{675} \qquad (\because b^3 < 2^{1009})$
- assuming H() = SHA-1(), max length of GARBAGE is:
  - $1024/8 - (2 + 8 + 1 + 35) = 82$ bytes $= 656$ bits
- roughly 19 bits less than $b^3 - (b-1)^3$, so around 2^{19} trials to find a working S'

Over-permissiveness in Openswan 2.6.50

`/** all numbers below are hexadecimals **/`

- Use this r’: `00 01` `GARBAGE` `00 30 21 … … 04 16 SHA-1(m’)`
- Want: $(a + b)^3 = a^3 + 3a^2b + 3b^2a + b^3$
- Finding 'a' is similar to the original attack algorithm
- construct t = 00 01 00 00 .. 00
  - compute $\alpha = \text{ceil}\left(\sqrt[3]{t}\right)$
  - sequentially find the largest 'c' s.t. $((\alpha / 2^c + 1)\ 2^c)^3$ match r’ before GARBAGE
  - then $a = (\alpha / 2^c + 1)\ 2^c$
    - this is to make as many LSBs of 'a' zeros
      - to avoid overlapping terms

Over-permissiveness in Openswan 2.6.50

- Use this r’: `/** all numbers below are hexadecimals **/` `00 01` `GARBAGE` `00 30 21 … … 04 16 SHA-1(m’)`
- Want: $(a + b)^3 = a^3 + 3a^2b + 3b^2a + b^3$
- Finding 'b' is a little more complex
- let r_b = `00 30 21 … … 04 16 SHA-1(m’)`, and $n' = 2^{|r_b|}$
  - r_b can be considered as (b^3 mod n’)
  - since n’ is a power of 2, $\phi(n') = 2^{|r_b|-1}$
  - we can guarantee r_b and n’ are coprime w/ an odd SHA-1(m’)
  - use Extended Euclidean Algorithm to find f, s.t.
    - $ef = 1 \pmod{2^{|r_b|-1}}$
      that is, f is the multiplicative inverse of e mod $\phi(n')$
  - finally $b = r_b^f \bmod n'$

Leniency in strongSwan 5.6.3

3. Accepting less than 8 bytes of padding
    - Can be used to make the other attacks easier
        - Use no padding, gain more bytes for GARBAGE

Leniency in axTLS 2.1.3

2. Ignoring prefix bytes

```
i = 10;
/* start at the first possible non-padded byte */
while (block[i++] && i < sig_len);
size = sig_len - i;
/* get only the bit we want */
if (size > 0) {... ...}
```

- First 10 bytes are not checked at all

Leniency in axTLS 2.1.3

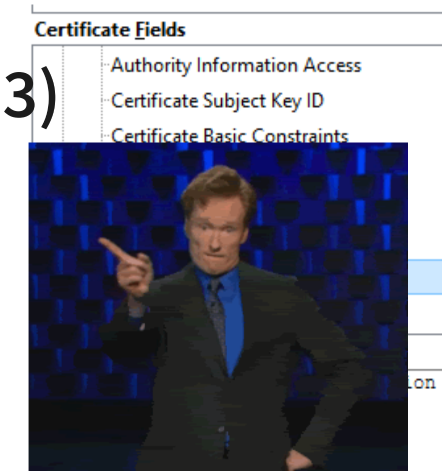
2. Ignoring prefix bytes
   - First 10 bytes directly skipped
   - Make forgery easier, use this r’ (first 90 bits are all zeros)

```
/** all numbers below are hexadecimals **/
00 00 00 00 00 00 00 00 00 00 00
30 21 … … 04 16 SHA-1(m’) GARBAGE
```

   - Reduce the distance between two consecutive perfect cubes
     - Easier to find S’

Leniency in axTLS 2.1.3

3. Ignoring AS.AlgorithmIdentifier (CVE-2018-16253)



- Just because H() is known from outside
- Doesn't mean it does not need to be checked
- This can be exploit together with the previous 2 flaws
- Use this r'

```
/** all numbers below are hexadecimals **/
00 00 00 00 00 00 00 00 00 00 00
30 00 30 00 04 H().size H(m') GARBAGE++
```

  - shorten the AlgorithmIdentifier to allow more **GARBAGE**
  - useful with shorter modulus and longer hashes

Leniency in axTLS 2.1.3

4. Trusting the declared ASN.1 DER lengths w/o sanity checks [CVE-2018-16149]
    - DoS PoC: making z exceptionally large crashed the verifier

- Particularly damaging
    - axTLS does certificate chain validation bottom-up
    - even if no small e in the wild
        - any MITM can inject a fake certificate with e = 3
        - crash verifier before the whole chain is verified against some trusted root anchors

Leniency in MatrixSSL 3.9.1 (CRL)

1. Mishandling Algorithm OID

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w
  /* [AlgorithmIdentifier] */
  30 x
    06 u 2B 0E 03 02 1A
    05 y
  /* [Digest] */
  04 z
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

can take arbitraily any values

- Some bytes in the middle of AS can take any values
    - Depends on choice of H(), SHA-1: 5 bytes, SHA-256: 9 bytes
- Doesn’t seem to be numerous enough for practical attacks

Discussion

- Parsing is hard!
- What robustness means depends on the context
  - Parser robustness (e.g. handling malformed inputs) is bad for security-critical scenarios
- Better way: construction-based verification
  - Many libraries have code for signing
    - for verifiers, instead of parsing,
      compute H(m), prepare AS and construct r_v
    - then see if $r_v == S^e \bmod n$