black hat USA 2019

AUGUST 3-8, 2019
MANDALAY BAY / LAS VEGAS

A Decade After Bleichenbacher '06, RSA Signature Forgery Still Works

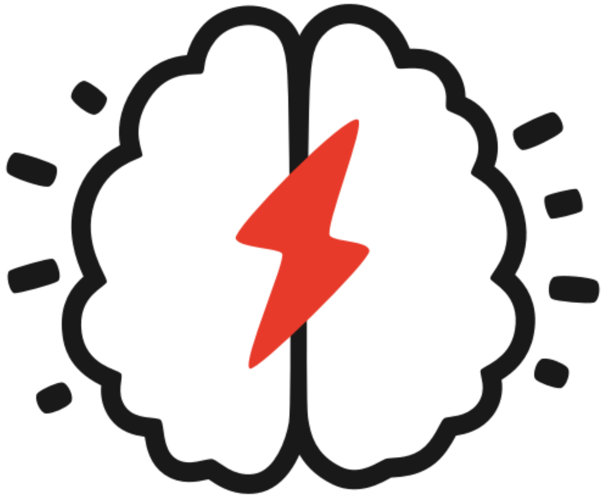
Sze Yiu Chau Purdue University
schau@purdue.edu

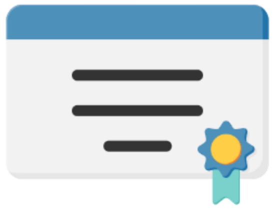
Who am I?

- Born and raised in Hong Kong

- PhD in CS from Purdue
- Joining CUHK IE as AP in 2020
- Interests: (in)secure design and implementations of protocols

A little brain teaser





What is common among these protocols?

THEY ALL HAVE RFCS

THEY'RE ALL SECURITY-CRITICAL

THEY ALL CAN BENEFIT FROM PKCS#1 V1.5 RSA SIGNATURES

Textbook RSA signature

- Signing message m:

m

H(m)

$H(m)^d \bmod n$

S

where d = private exponent
n = modulus

- 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

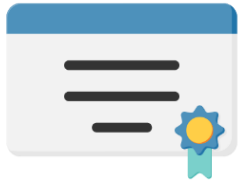
1. Which H() to use?
   - SHA-1, SHA-2 family, SHA-3 family …
2. n is usually much longer than H(m)
   - $|n| \geq 2048$-bit
   - $|\text{SHA-1}| = 160$-bit, $|\text{SHA-256}| = 256$-bit

- Need meta-data and padding

picture borrowed from *http://www.howtobeadad.com/2016/29188/reality-vision-for-parents*

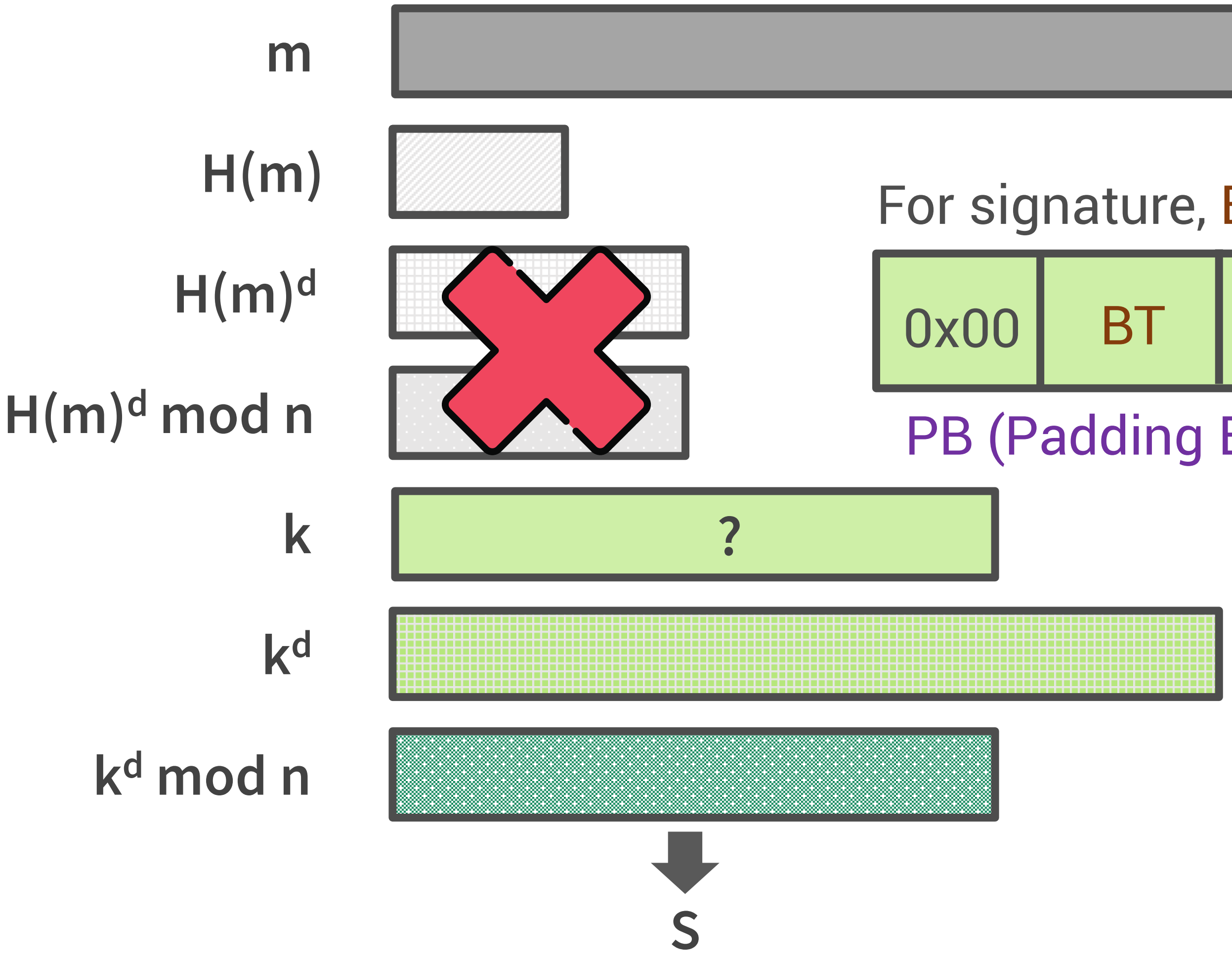
Beyond textbook RSA

- The PKCS#1 family of standards
- Both encryption and signature schemes
  - version 2+ adapted schemes from Bellare et al.
- For signatures, PKCS#1 v1.5 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

- Encoded AS looks like this:

```
30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14 2A AE 6C
35 C9 4F CF B4 15 DB E9 5F 40 8B 9C E9 1E E8 46 ED
```

  - H() = SHA-1(), m = “hello world”
  - altogether 35 bytes
- DER encoded object is a tree of <T,L,V> triplets

PKCS#1 v1.5 Signature Scheme

- Encoded AS looks like this:

```
T  L
30 21
      30 09 06 05 2B 0E 03 02 1A 05 00 04 14 2A AE 6C
    V 35 C9 4F CF B4 15 DB E9 5F 40 8B 9C E9 1E E8 46 ED
```

  - H() = SHA-1(), m = “hello world”
  - altogether 35 bytes
- DER encoded object is a tree of <T,L,V> triplets

PKCS#1 v1.5 Signature Scheme

- Encoded AS looks like this:

```
T  L
30 21 T  L
      30 09
         V 06 05 2B 0E 03 02 1A 05 00
      04 14 2A AE 6C 35 C9 4F CF B4 15
    V
      DB E9 5F 40 8B 9C E9 1E E8 46 ED
```

  - H() = SHA-1(), m = “hello world”
  - altogether 35 bytes
- DER encoded object is a tree of <T,L,V> triplets

PKCS#1 v1.5 Signature Scheme

- Encoded AS looks like this:

```
T  L
30 21 T  L
      30 09 T  L        V
            06 05 2B 0E 03 02 1A   <- OID of SHA-1
            05 00                  <- null parameter
   V  04 14 2A AE 6C 35 C9 4F CF B4 15
      DB E9 5F 40 8B 9C E9 1E E8 46 ED
```

  - H() = SHA-1(), m = “hello world”
  - altogether 35 bytes
- DER encoded object is a tree of <T,L,V> triplets

PKCS#1 v1.5 Signature Scheme

- Encoded AS looks like this:

```
T  L
30 21 T  L
      30 09 T  L        V
            06 05 2B 0E 03 02 1A   <- OID of SHA-1
      T  L  05 00                  <- null parameter
      04 14 2A AE 6C 35 C9 4F CF B4 15
      DB E9 5F 40 8B 9C E9 1E E8 46 ED   <- SHA-1("hello world")
                        V
```

  - H() = SHA-1(), m = “hello world”
  - altogether 35 bytes
- DER encoded object is a tree of <T,L,V> triplets

PKCS#1 v1.5 Signature Scheme

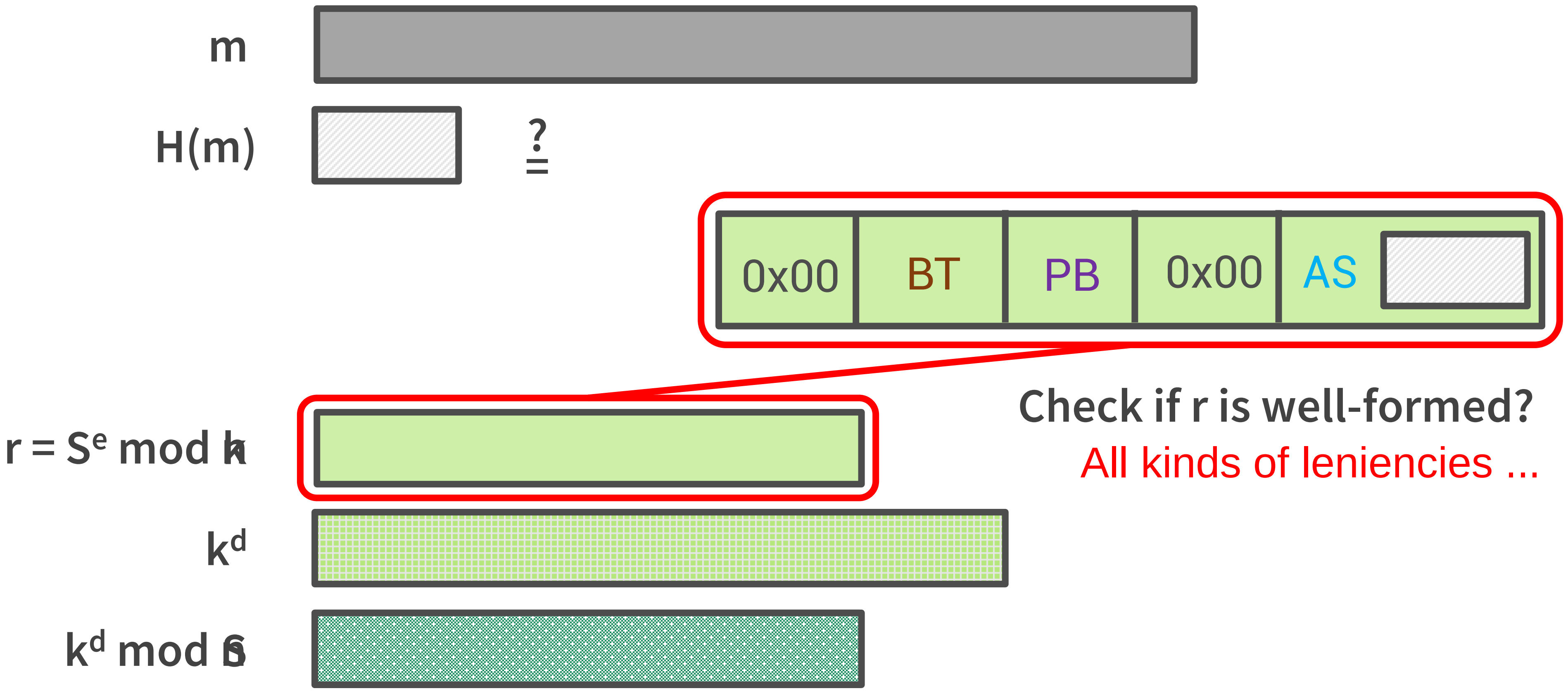
- Encoded AS looks like this:

```
T  L
30 21 T  L
      30 09 T  L           V
            06 05 2B 0E 03 02 1A
 +    T  L  05 00
      04 14 2A AE 6C 35 C9 4F CF B4 15
      DB E9 5F 40 8B 9C E9 1E E8 46 ED
                           V
```

  - H() = SHA-1(), m = “hello world”
  - altogether 35 bytes
- DER encoded object is a tree of <T,L,V> triplets

PKCS#1 v1.5 Signature Scheme

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



RSA and Factorization

Given (e,n) can we find d?

- $ed = 1 \pmod{\phi(n)}$
  d is the multiplicative inverse of e mod $\phi(n)$
- if we know
  $\phi(n) = (p-1)(q-1)$
  then easy to find d
  (use Extended Euclidean Algorithm)
- if we ... = pq
  then e... find $\phi(n)$

RSA-640 [edit]

RSA-640 has 640 bits (193 decimal digits). A cash prize of US$20,000 was offered by RSA Security for a successful factorization. On November 2, 2005, F. Bahr, M. Boehm, J. Franke and T. Kleinjung of the German Federal Office for Information Security announced that they had factorized the number using GNFS as follows:[25][26][27]

```
RSA-640 = 31074182404900437213507500358885679300373460228427275457
          20161948823206440518081504556346829671723286782437916272
          83803341547107310850191954852900733772482278352574238645
          4014691736602477652346609
```

```
RSA-640 = 16347336458092538484431338838650908598417836700330923121
          811108523893331001045081512121181675l1579
        × 19008712816648221131268515739354139754718967899685154936
          666385390880271038021044989571912614655 71
```

The computation took five months on 80 2.2 GHz AMD Opteron CPUs.

The slightly larger RSA-200 was factored in May 2005 by the same team.

RSA-200 [edit]

RSA-200 has 200 decimal digits (663 bits), and factors into the two 100-digit primes given below.

Wikinews has related news: *Two hundred digit number factored*

On May 9, 2005, F. Bahr, M. Boehm, J. Franke, and T. Kleinjung announced[28][29] that they had factorized the number using GNFS as follows:

```
RSA-200 = 27997833911221327870829467638722601621070446786955428537560009929326128400107609345671052955360856061822351910951365788637105954482006576775098580557613579098734950144178863178946295187237869221823983
```

```
RSA-200 = 3532461934402770121272604978198464368671197400197625023649303468776121253679423200058547956528088349
        × 7925869954478333033347085841480059687737975857364219960734330341455767872818152135381409304740185467
```

The CPU time spent on finding these factors by a collection of parallel computers amounted – very approximately – to the equivalent of 75 years work for a single 2.2 GHz Opteron-based computer.[28] Note that while this approximation serves to suggest the scale of the effort, it leaves out many complicating factors; the announcement states it more precisely.

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 some m without d

Bleichenbacher's low exponent attack

- A contributing factor to the push for bigger e (e.g. 65537)

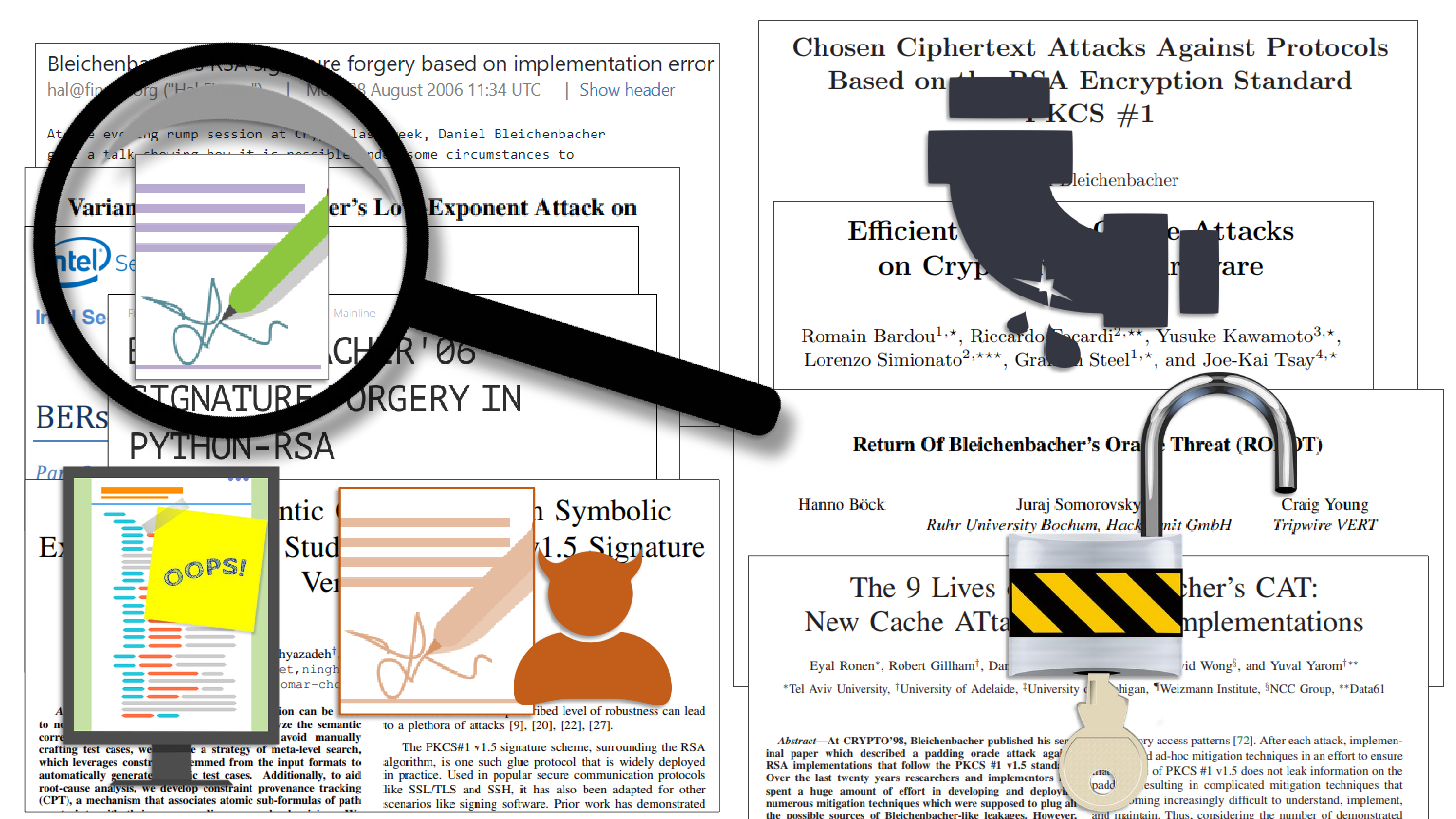
- Smaller e more efficient for signature verifier
  - e = 3 prescribed in some protocols

    e.g. DNSSEC [RFC3110, Sect. 4]

4. Performance Considerations

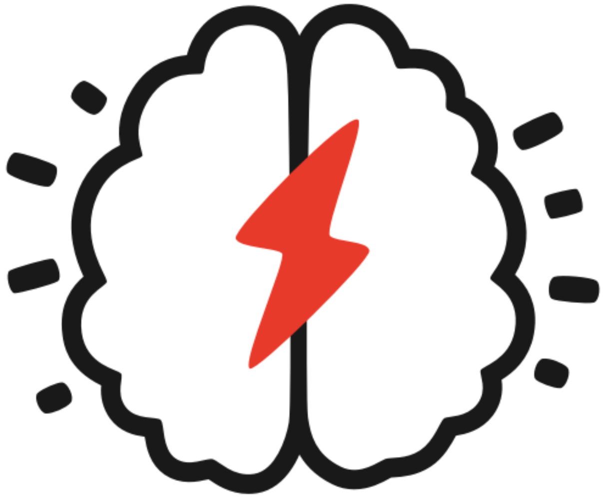
General signature generation speeds are roughly the same for RSA and DSA [RFC2536]. With sufficient pre-computation, signature generation with DSA is faster than RSA. Key generation is also faster for DSA. However, signature verification is an order of magnitude slower with DSA when the RSA public exponent is chosen to be small as is recommended for KEY RRs used in domain name system (DNS) data authentication.

A public exponent of 3 minimizes the effort needed to verify a signature. Use of 3 as the public exponent is weak for confidentiality uses since, if the same data can be collected encrypted under three different keys with an exponent of 3 then, using the Chinese Remainder Theorem [NETSEC], the original plain text can be easily recovered. If a key is known to be used only for authentication, as is the case with DNSSEC, then an exponent of 3 is acceptable. However other applications in the future may wish to leverage DNS distributed keys for applications that do require confidentiality. For keys which might have such other uses, a more conservative choice would be 65537 (F4, the fourth fermat number).



Bleichenba
re forgery based on implementation error
hal@fin
8 August 2006 11:34 UTC
Show header
At
eek, Daniel Bleichenbacher
some circumstances to
Varian
er's Lo
Exponent Attack on
Se
In
Se
Mainline
B
CHER'06
SIGNATURE
ORGERY IN
PYTHON-RSA
BERs
OOPS!
ntic
Symbolic
Stud
v1.5 Signature
Ve
hyazadeh
et,ningh
omar-cho
ibed level of robustness can lead to a plethora of attacks [9], [20], [22], [27].
The PKCS#1 v1.5 signature scheme, surrounding the RSA algorithm, is one such glue protocol that is widely deployed in practice. Used in popular secure communication protocols like SSL/TLS and SSH, it has also been adapted for other scenarios like signing software. Prior work has demonstrated
crafting test cases, we
e a strategy of meta-level search,
which leverages constr
emmed from the input formats to
automatically generate
c test cases. Additionally, to aid
root-cause analysis, we develop constraint provenance tracking
(CPT), a mechanism that associates atomic sub-formulas of path
Chosen Ciphertext Attacks Against Protocols
Based on
A Encryption Standard
KCS #1
Bleichenbacher
Efficient
e Attacks
on Cryp
are
Romain Bardou[1,*], Riccardo
ocardi[2,**], Yusuke Kawamoto[3,*],
Lorenzo Simionato[2,***], Gra
Steel[1,*], and Joe-Kai Tsay[4,*]
Return Of Bleichenbacher's Ora
Threat (RO
T)
Hanno Böck
Juraj Somorovsky
Craig Young
Ruhr University Bochum, Hack
nit GmbH
Tripwire VERT
The 9 Lives
her's CAT:
New Cache ATta
mplementations
Eyal Ronen*, Robert Gillham†, Da
vid Wong§, and Yuval Yarom†**
*Tel Aviv University, †University of Adelaide, ‡University
higan, ¶Weizmann Institute, §NCC Group, **Data61
Abstract—At CRYPTO'98, Bleichenbacher published his se
inal paper which described a padding oracle attack aga
RSA implementations that follow the PKCS #1 v1.5 stand
Over the last twenty years researchers and implementors
spent a huge amount of effort in developing and deploy
numerous mitigation techniques which were supposed to plug al
ory access patterns [72]. After each attack, implemen-
d ad-hoc mitigation techniques in an effort to ensure
of PKCS #1 v1.5 does not leak information on the
esulting in complicated mitigation techniques that
oming increasingly difficult to understand, implement,

A little brain teaser



What is common among these protocols?

THEY ALL HAVE RFCS

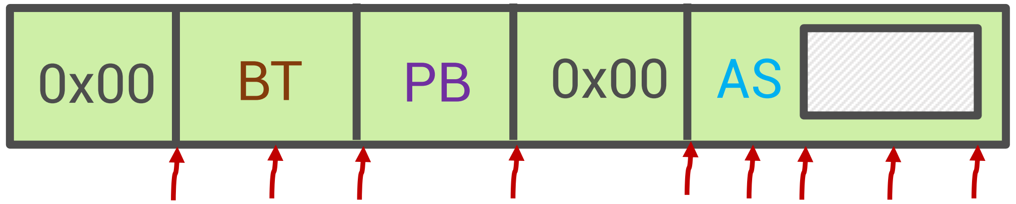
THEY'RE ALL SECURITY-CRITICAL

SUFFER

THEY ALL CAN ~~BENEFIT~~ FROM PKCS#1 V1.5 RSA SIGNATURES

Why was the attack possible?

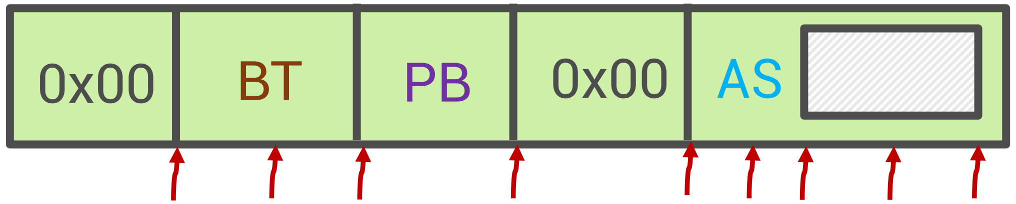
- Problem: verifier accept malformed input w/ GARBAGE unchecked
  - Can be in many different locations, not only at the end



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

To find these attacks

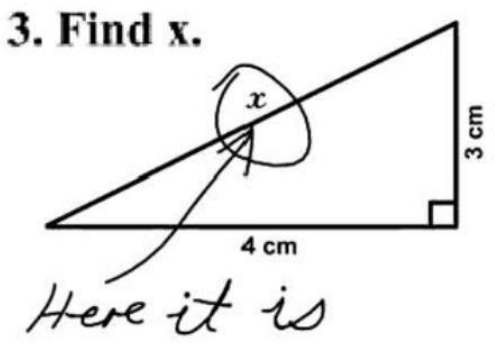
- Want to see how input bytes are being checked

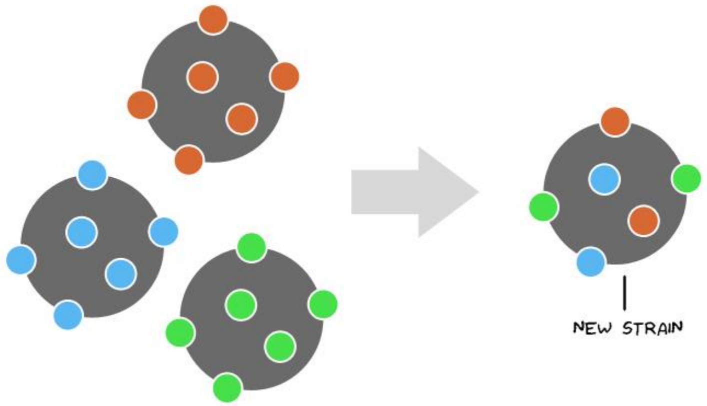


- If enough unchecked GARBAGE then

Automatically generate concolic test cases

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



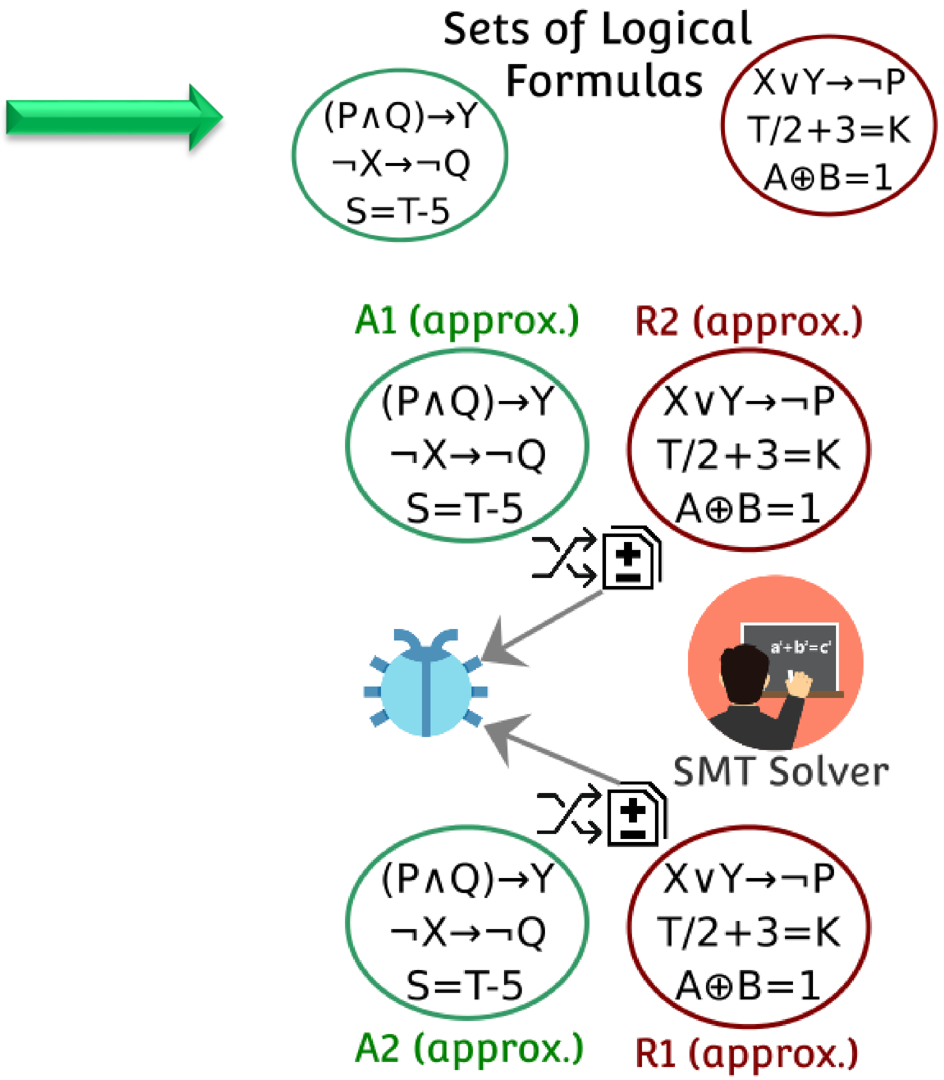


- Based on that, automatically pack symbolic/concrete components into test buffers

pictures borrowed from *https://www.memecenter.com/fun/160/Find-x* and *https://askascientist.co.uk/health/viruses-mutate/*

Testing with Symbolic Execution

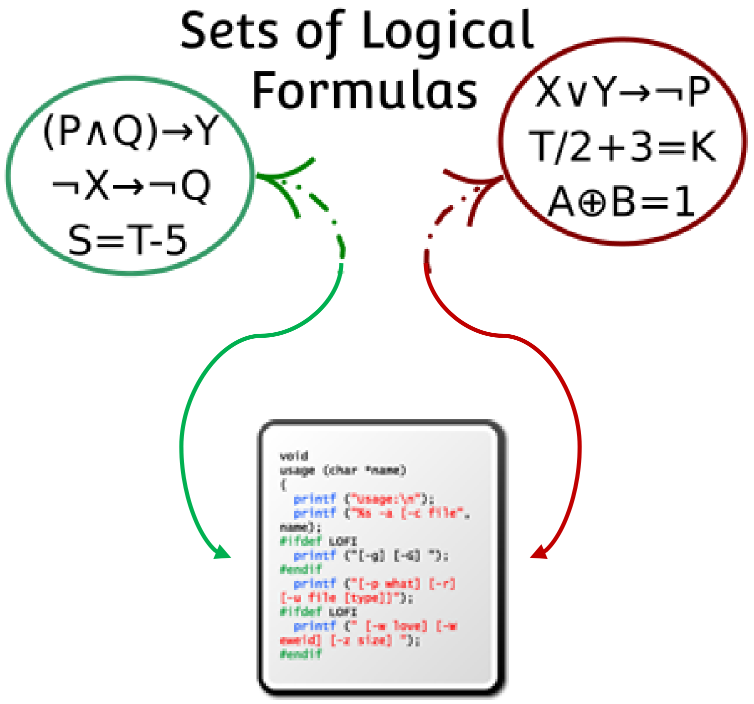
- Symbolic Execution with concolic test cases



- 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>`
  - Function filtering, e.g. `memcmp()`
  - Tiny space & time overhead



Implementations Tested

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	-

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

* 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
- 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 $S' = (a+b)$

```
/* 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;
```

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 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
```

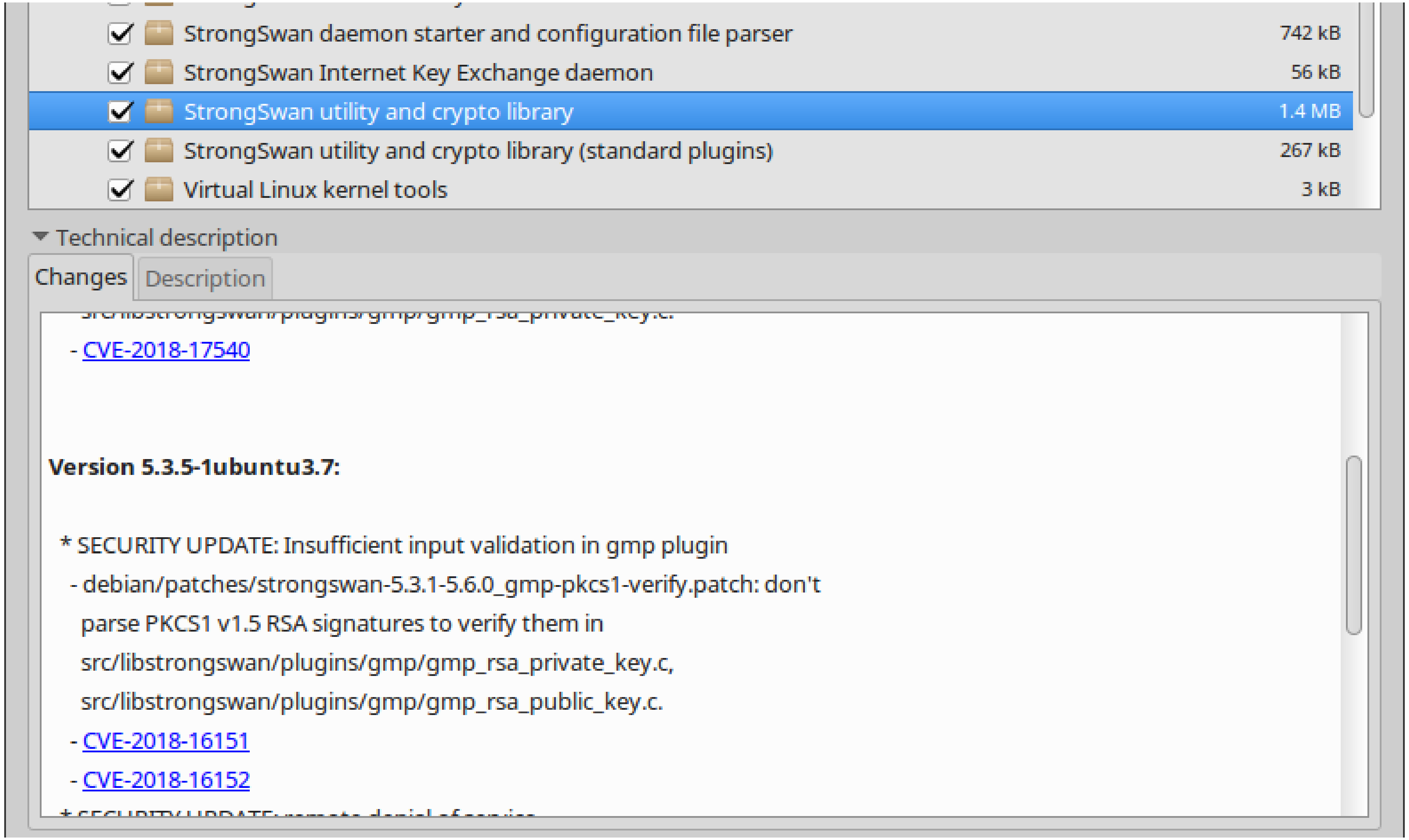
both would be recognized as OID of SHA-1

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

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

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 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 cubes
     - Easier to find S'

- roughly 19 bits < b^3 $)^3$, so ~ 2^{19} trials to find S'

Leniency in axTLS 2.1.3

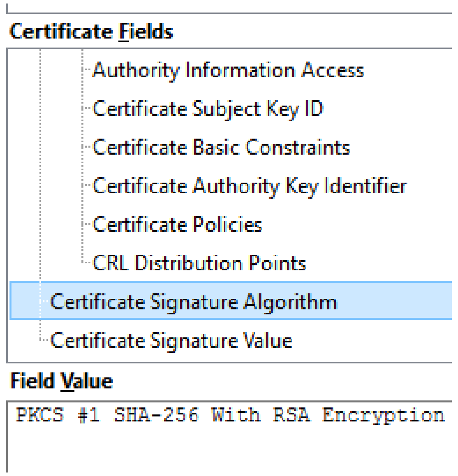
3. Ignoring AS.AlgorithmIdentifier [CVE-2018-16253]

```
/** 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 ...

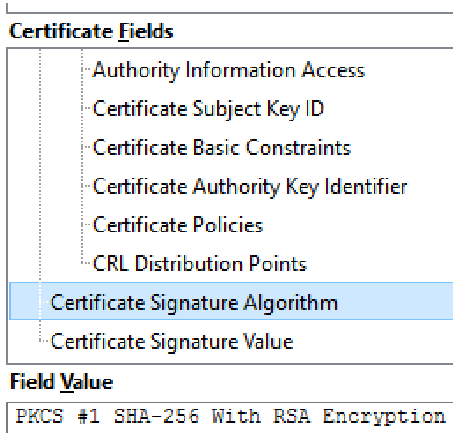
```
if (asn1_next_obj(asn1_sig, &offset,
    ASN1_SEQUENCE) < 0 ||
    asn1_skip_obj(asn1_sig, &offset,
    ASN1_SEQUENCE))      goto end_get_sig;

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

3. 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

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

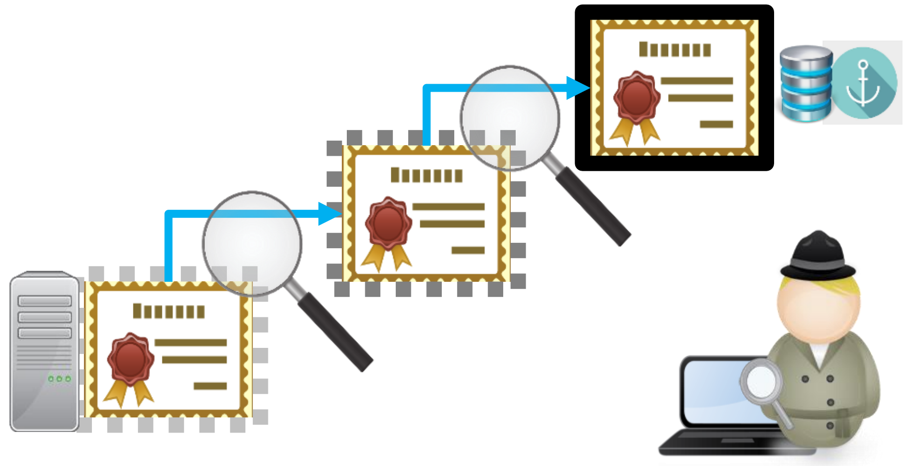
```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w
  /* [AlgorithmIdentifier] */
  30 x
    06 u 2B 0E 03 02 1A
    05 y
  /* [Digest] */
  04 z
    /* H(m), H()=SHA-1(), m = "hello world" */
    2A AE 6C 35 C9 4F CF B4 15 DB
    E9 5F 40 8B 9C E9 1E E8 46 ED
```

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

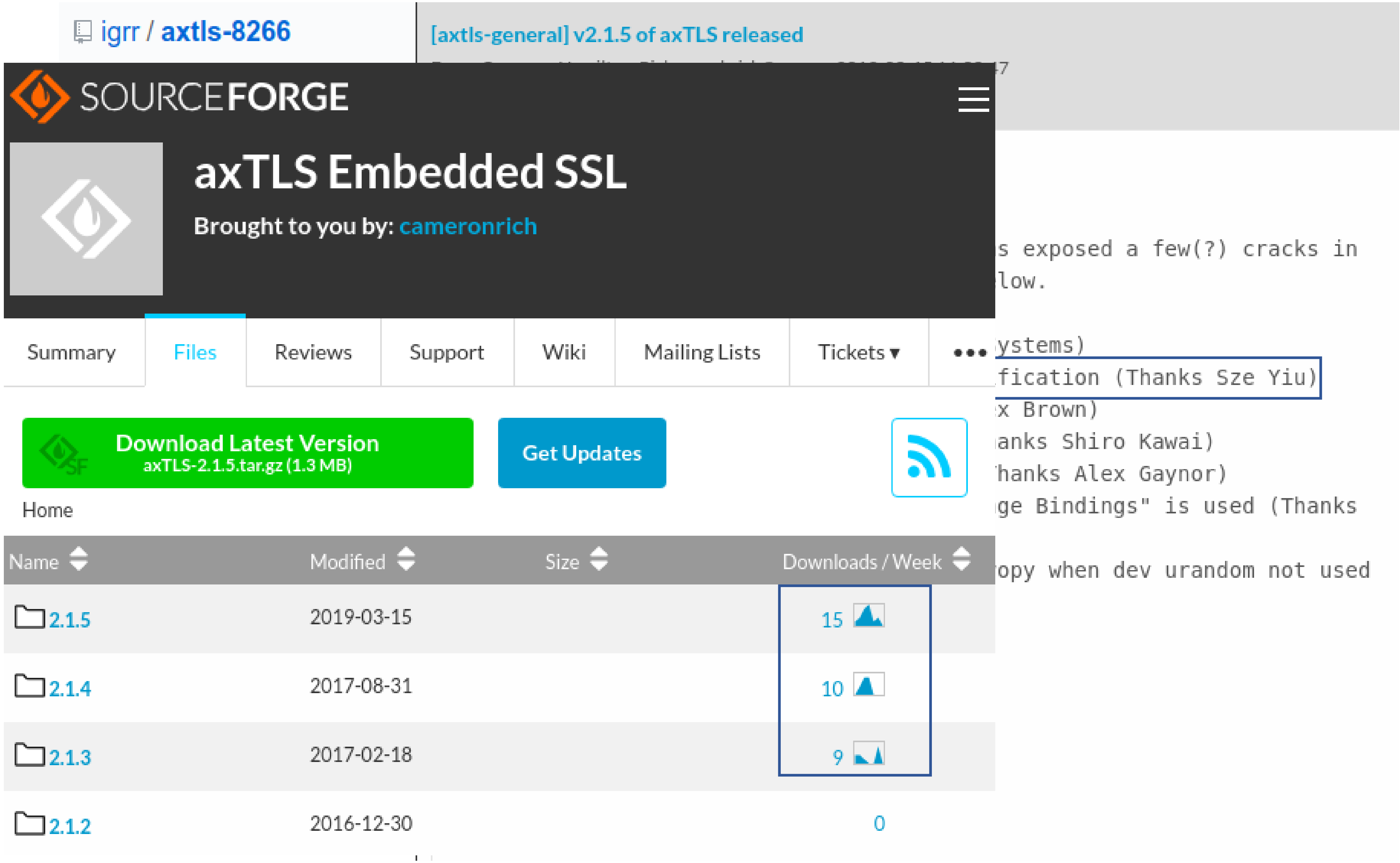
- DoS PoC: making z exceptionally large crashed the verifier

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 during chain traversal



patching axTLS



Leniency in libtomcrypt 1.16

1. Accepting trailing GARBAGE
   - original Bleichenbacher '06 forgery also works

2. Accepting less than 8 bytes of padding
   - Use no padding, gain more bytes for GARBAGE
     - Make signature forgery easier

- *Flaws independently found by other researchers, fixed in v1.18*

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

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.

Summary

- 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 some other unwarranted leniencies

Lessons Learned

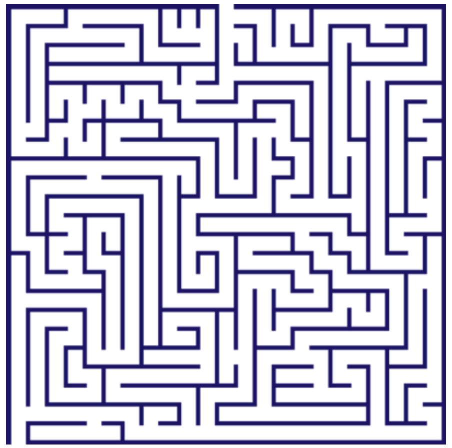
- Corner-cutting is not cool

 $\Rightarrow$  

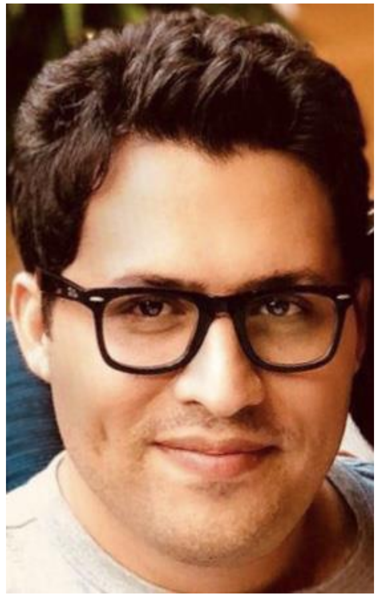
- Learn from previous mistakes

- Parsing is hard

Moosa Yahyazadeh

The University of Iowa

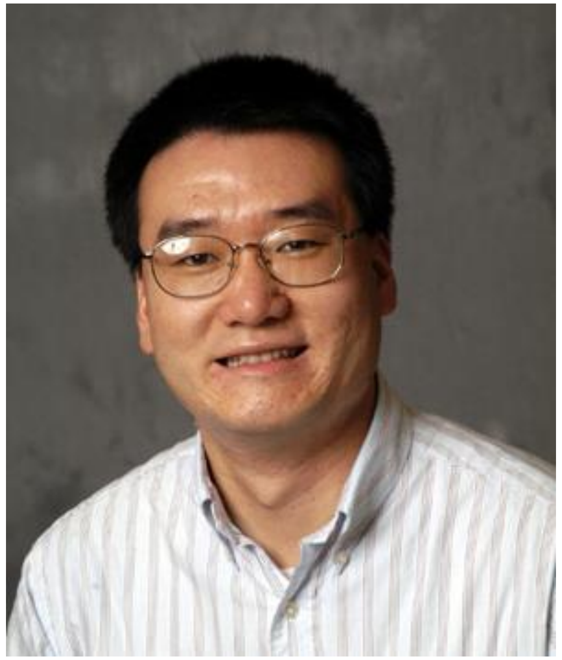


Omar Chowdhury

The University of Iowa

Aniket Kate

Purdue University



Ninghui Li

Purdue University