

CSE 127: Introduction to Security

Lecture 16: Side Channels and
Constant-Time Code

Nadia Heninger and Deian Stefan

UCSD

Fall 2019

Some material from Dan Boneh, Stefan Savage

Reminder: Side-channel attacks

You saw before how timing information (from caches or implementation choices) could leak secret information from a running program.

This lecture:

- A variety of different side-channel attacks
- How side-channel attacks can be used against cryptography
- How to mitigate timing side channels in code

Different types of side channels

Computers are physical objects, so measuring them during program execution can reveal information about the program or data.

- Electromagnetic radiation
 - Voltage running through a wire produces a magnetic field
- Power consumption
 - Different paths through a circuit might consume different amounts of power
- Sound (acoustic attacks)
 - Capacitors discharging can make noises
- Timing
 - Different execution time due to program branches
 - Cache timing attacks
- Error messages
 - Error messages might reveal secret information to an attacker
- Fault attacks

TEMPEST/van Eck Phreaking

"Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk?" Wim van Eck 1985

- Governments knew about emissions for decades (TEMPEST)
- Surprising that it could be done with off the shelf equipment

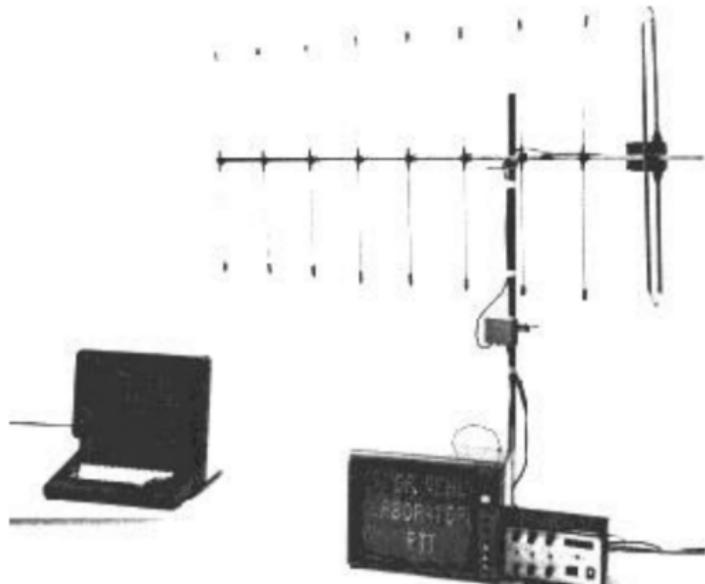


Fig. 1. Eavesdropping set-up using a variable oscillator and a frequency divider to restore synchronization. The picture on the TV is picked up from the radiation of the VDU in the background.

"Electromagnetic Eavesdropping Risks of Flat-Panel Displays" Kuhn 2004

- Image displays simultaneously along line
- Pick up radiation from screen connection cable

350 MHz, 50 MHz BW, 12 frames (160 ms) averaged

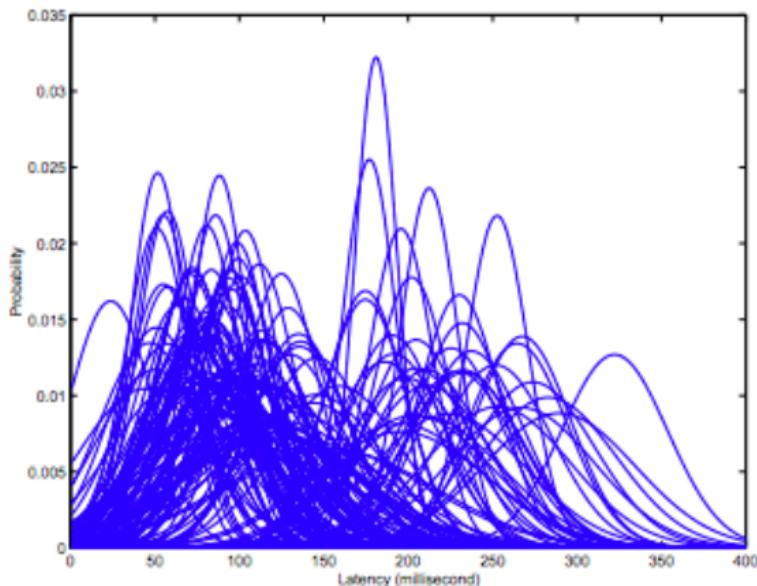


US/NATO define TEMPEST shielding standards



"Timing Analysis of Keystrokes and Timing Attacks on SSH" Song Wagner Tian 2001

- In interactive SSH, keystrokes sent in individual packets
- Build model of inter-keystroke delays by finger, key pair
- Measure packet timing off network, do Viterbi decoding



Side-Channel Attacks and Cryptography

Traditional security models for cryptography focus on indistinguishability of ciphertexts, adversaries who can request encryption or decryption oracles.

Cryptographic program execution can leak information about secrets.

Outside of traditional security models for cryptography.

Timing Attacks on Modular Exponentiation

Kocher 96

RSA performs modular exponentiation: $m = c^d \bmod N$

Pseudocode for “square and multiply” modular exponentiation algorithm:

```
m = 1
for i = 0 ... len(d):
    if d[i] = 1:
        m = c * m mod N
    m = square(m) mod N
return m
```

- Number of multiplications performed leaks Hamming weight of private key
- Secret-dependent program execution time
- Turn into full attack by cleverly choosing ciphertexts

Power Analysis Attacks on Modular Exponentiation

Kocher Jaffe Jun 1998

Simple power analysis attacks plot power consumption over time.

The textbook square and multiply implementation clearly leaks secret key bits in a power trace.

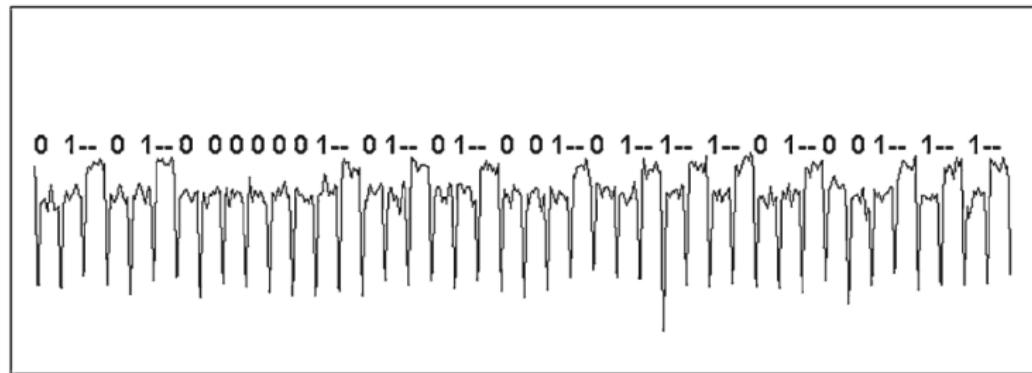
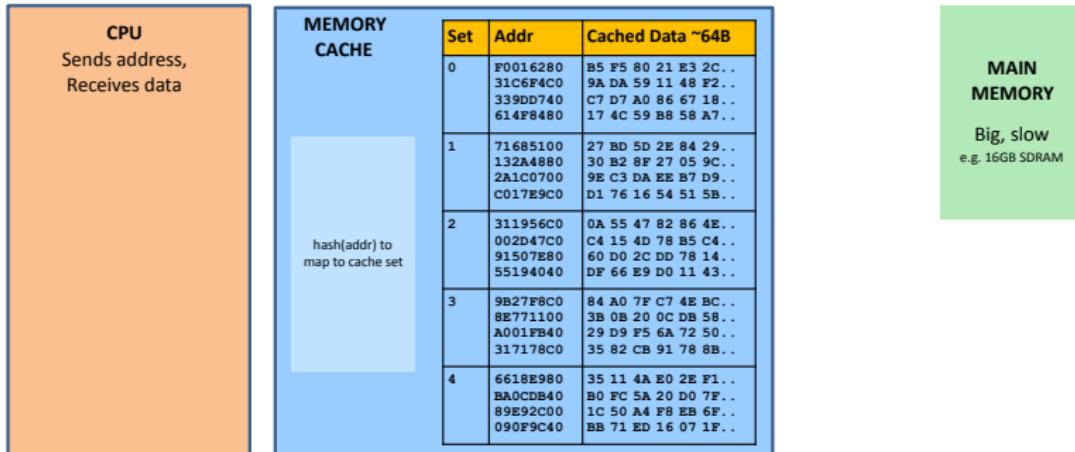


Fig. 11 SPA leaks from an RSA implementation

Reminder: Memory caches and cache attacks

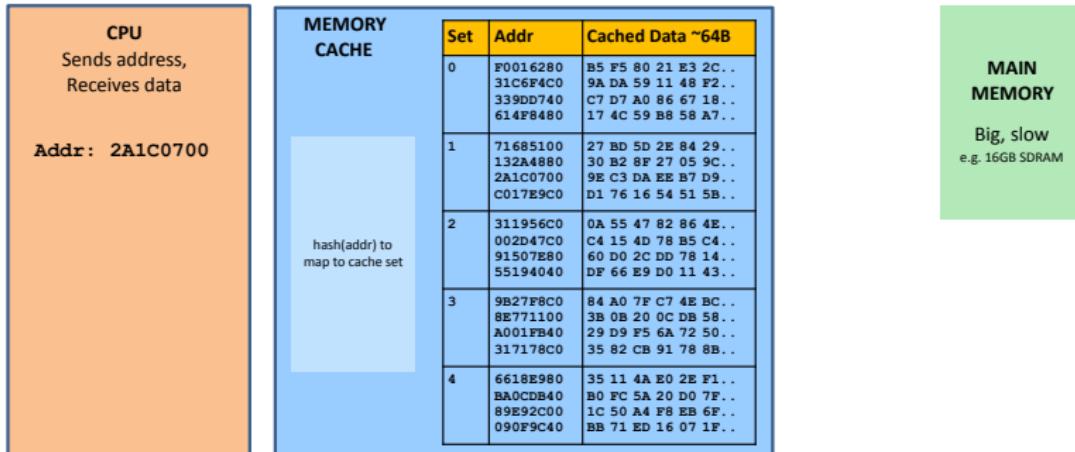
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

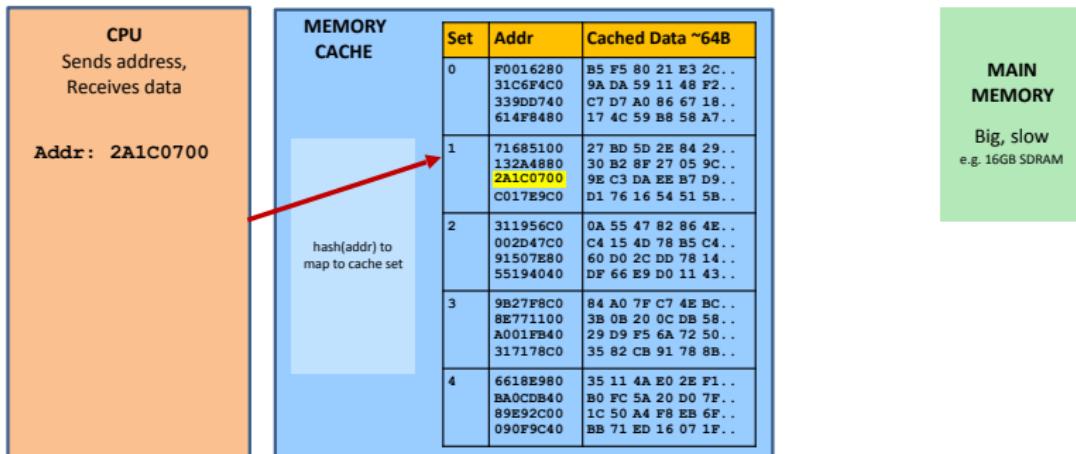
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

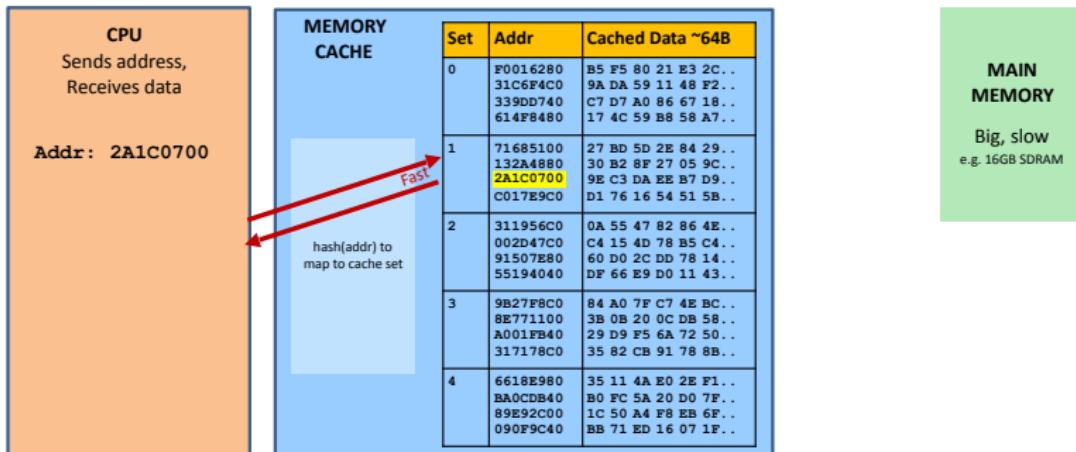
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

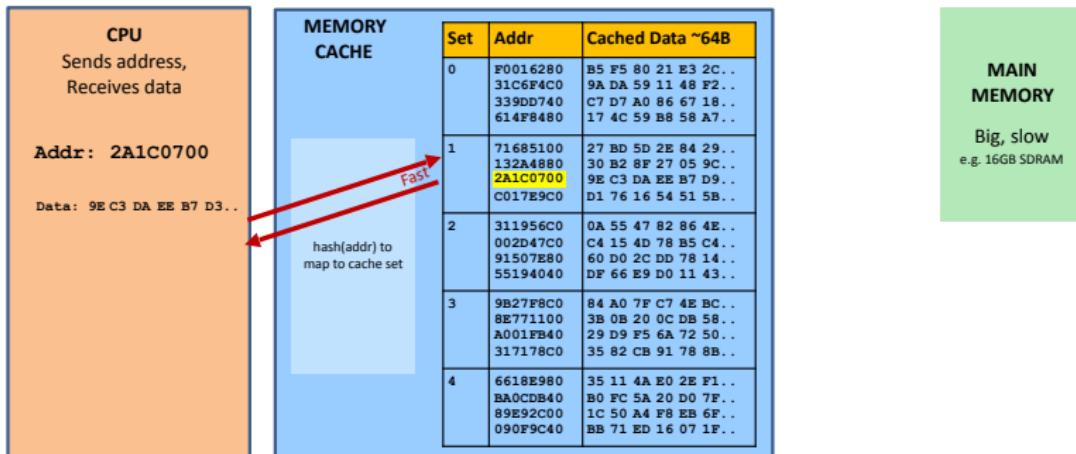
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

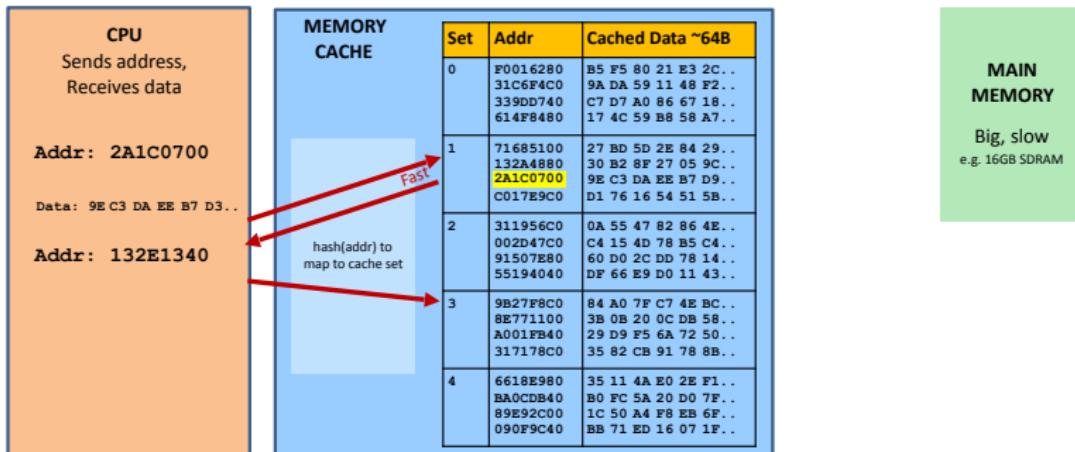
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

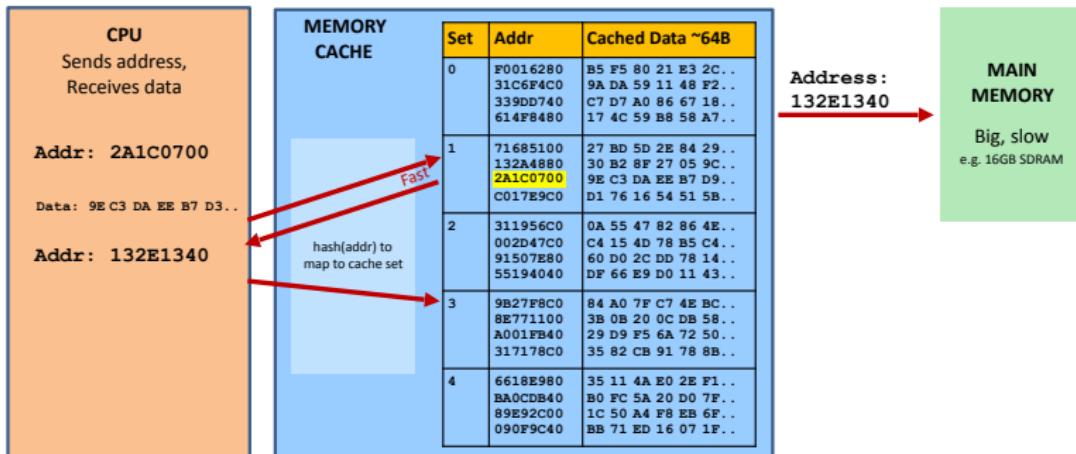
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

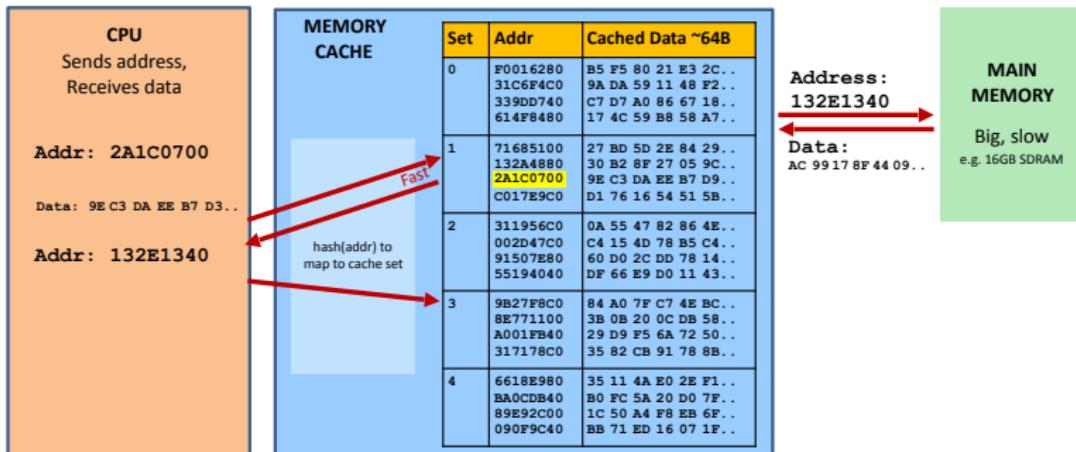
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

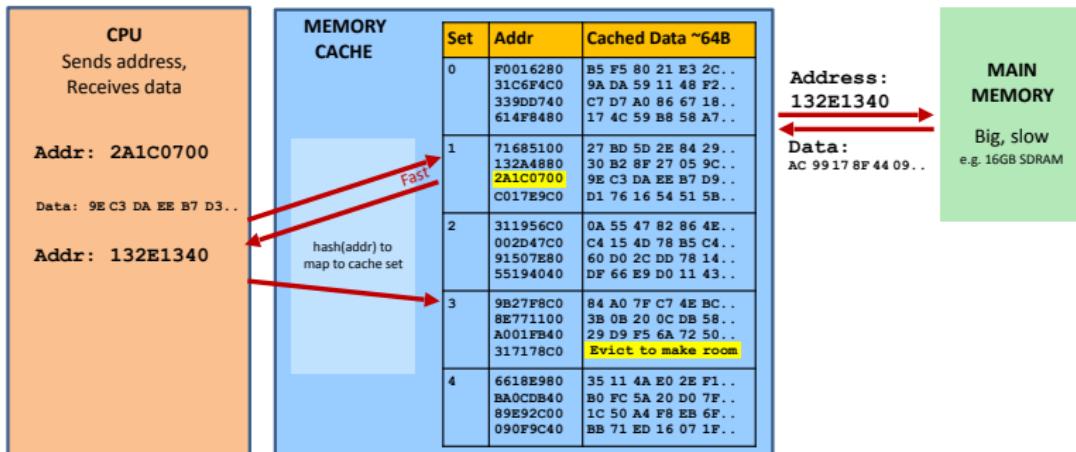
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

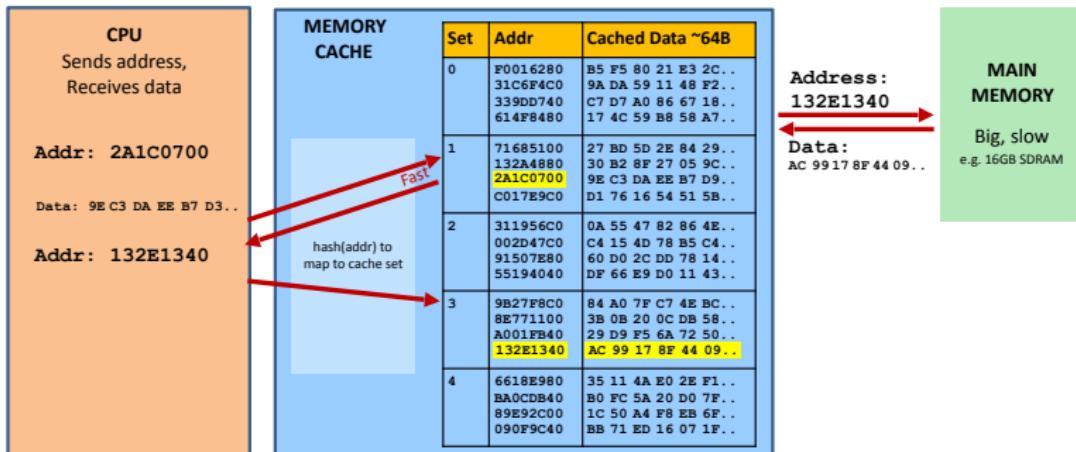
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

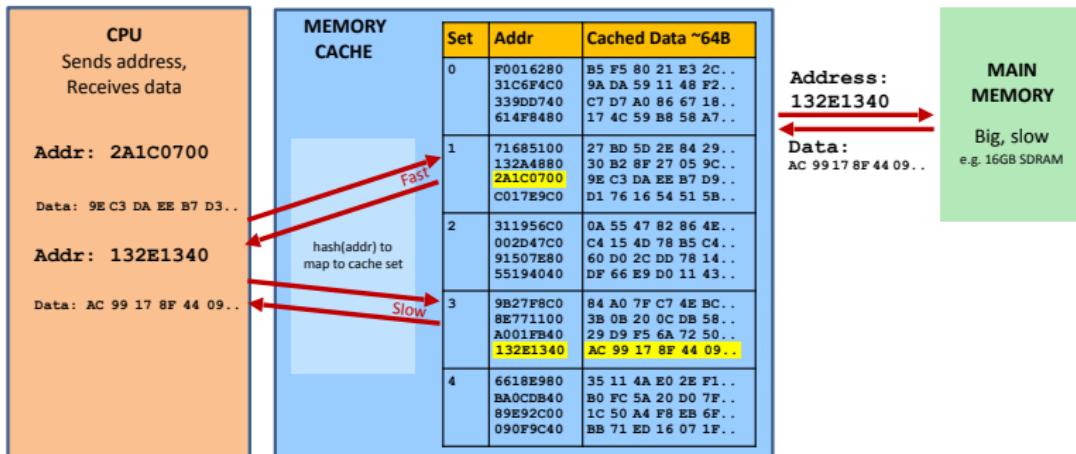
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

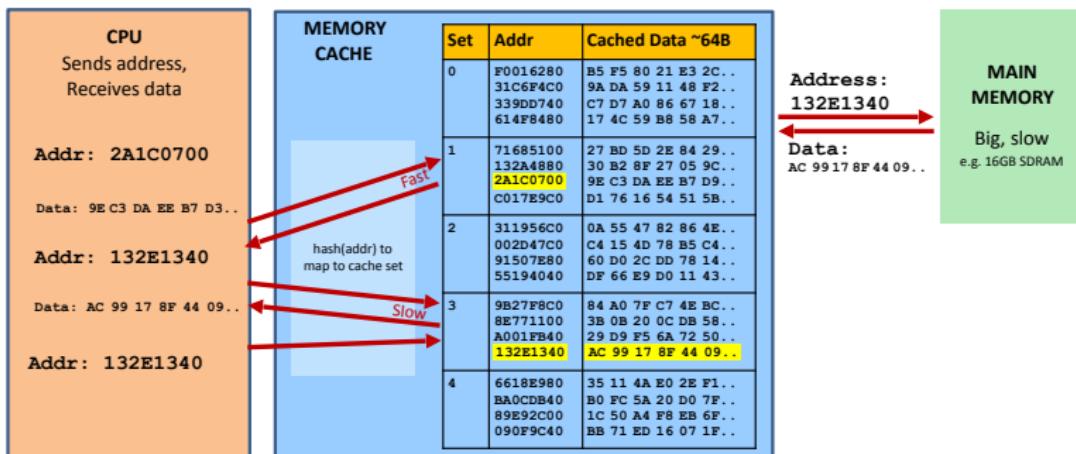
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

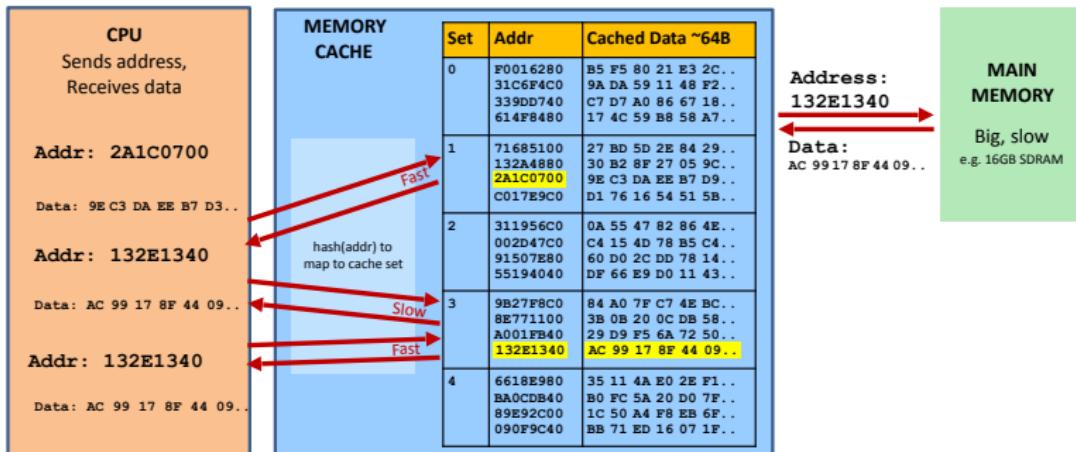
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

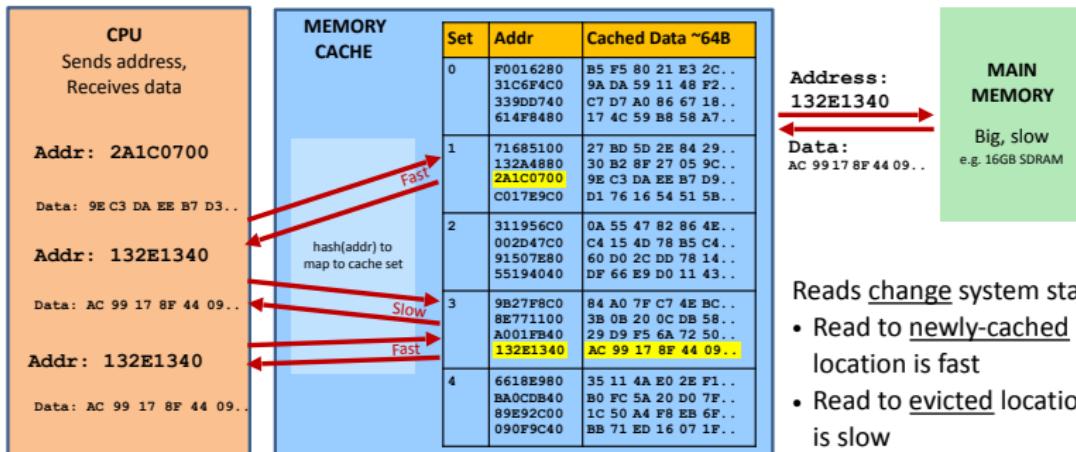
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Reminder: Memory caches and cache attacks

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory

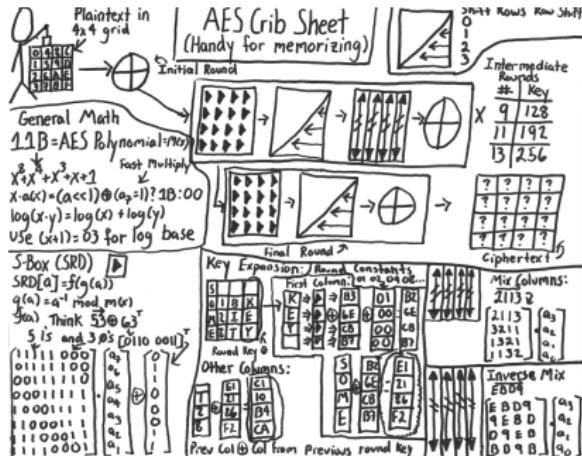


- In a cache attack, an attack program runs on the same processor as a victim program.
- The attack program measures memory access times to determine which data the victim loaded into cache.

Cache Attacks against AES

Bernstein 2005

- AES algorithm consists of xors, shifts, substitutions
- For speed, operations precomputed as a lookup table
- Table queries dependent on key values
- A cache attack can reveal the lookup locations and thus the secret key



Speculative Execution

- CPUs can guess likely program path and do speculative execution
- Example

```
if (uncached_value == 1) // load from memory  
    a = compute(b)
```

- Branch predictor guesses if() is true based on prior history
- Starts executing compute(b) speculatively
- When value arrives from memory, check if guess was correct:
 - Correct: Save speculative work → performance gain
 - Incorrect: Discard speculative work → no harm?

Spectre and Meltdown

Lipp et al., Kocher et al. 2017

Misspeculation

- Exceptions and incorrect branch prediction can cause “rollback” of transient instructions
- Old register states are preserved, can be restored
- Memory writes are buffered, can be discarded
- Cache modifications are not restored



- Spectre and Meltdown carry out cache attacks against speculatively loaded data so that an unprivileged attacker process can read kernel memory, break ASLR, etc.

Spectre variant 1 attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Before attack:

- Train branch predictor to expect if() is true (e.g. call with `x < array1_size`)
- Evict `array1_size` and `array2[]` from cache

Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base:

8 bytes of data (value doesn't matter)

Memory at `array1` base+1000:

`09 F1 98 CC 90...` (something secret)

`array2[0*4096]`
`array2[1*4096]`
`array2[2*4096]`
`array2[3*4096]`
`array2[4*4096]`
`array2[5*4096]`
`array2[6*4096]`
`array2[7*4096]`
`array2[8*4096]`
`array2[9*4096]`
`array2[10*4096]`
`array2[11*4096]`

..

Contents don't matter
only care about cache **status**

Uncached

Cached

Spectre variant 1 attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Memory & Cache Status

array1_size = 00000008

Memory at array1 base:

8 bytes of data (value doesn't matter)

Memory at array1 base+1000:

09 F1 98 CC 90... (something secret)

array2[0*4096]
array2[1*4096]
array2[2*4096]
array2[3*4096]
array2[4*4096]
array2[5*4096]
array2[6*4096]
array2[7*4096]
array2[8*4096]
array2[9*4096]
array2[10*4096]
array2[11*4096]

..

Contents don't matter
only care about cache **status**

Uncached

Cached

Spectre variant 1 attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Memory & Cache Status

array1_size = 00000008

Memory at array1 base:

8 bytes of data (value doesn't matter)

Memory at array1 base+1000:

09 F1 98 CC 90... (something secret)

array2[0*4096]
array2[1*4096]
array2[2*4096]
array2[3*4096]
array2[4*4096]
array2[5*4096]
array2[6*4096]
array2[7*4096]
array2[8*4096]
array2[9*4096]
array2[10*4096]
array2[11*4096]

..

Contents don't matter
only care about cache **status**

Uncached

Cached

Spectre variant 1 attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Speculative exec while waiting for array1_size:

Memory & Cache Status

array1_size = 00000008

Memory at array1 base:

8 bytes of data (value doesn't matter)

Memory at array1 base+1000:

09 F1 98 CC 90... (something secret)

array2[0*4096]
array2[1*4096]
array2[2*4096]
array2[3*4096]
array2[4*4096]
array2[5*4096]
array2[6*4096]
array2[7*4096]
array2[8*4096]
array2[9*4096]
array2[10*4096]
array2[11*4096]

..

Contents don't matter
only care about cache **status**

Uncached

Cached

Spectre variant 1 attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Speculative exec while waiting for array1_size:

- Predict that if() is true

Memory & Cache Status

array1_size = 00000008

Memory at array1 base:

8 bytes of data (value doesn't matter)

Memory at array1 base+1000:

09 F1 98 CC 90... (something secret)

array2[0*4096]
array2[1*4096]
array2[2*4096]
array2[3*4096]
array2[4*4096]
array2[5*4096]
array2[6*4096]
array2[7*4096]
array2[8*4096]
array2[9*4096]
array2[10*4096]
array2[11*4096]

..

Contents don't matter
only care about cache **status**

Uncached

Cached

Spectre variant 1 attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Speculative exec while waiting for array1_size:

- Predict that if() is true
- Read address (array1 base + x)
(using out-of-bounds x=1000)

Memory & Cache Status

array1_size = 00000008

Memory at array1 base:

8 bytes of data (value doesn't matter)

Memory at array1 base+1000:

09 F1 98 CC 90... (something secret)

array2[0*4096]
array2[1*4096]
array2[2*4096]
array2[3*4096]
array2[4*4096]
array2[5*4096]
array2[6*4096]
array2[7*4096]
array2[8*4096]
array2[9*4096]
array2[10*4096]
array2[11*4096]

..

Contents don't matter
only care about cache **status**

Uncached

Cached

Spectre variant 1 attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Speculative exec while waiting for array1_size:

- Predict that if() is true
- Read address (array1 base + x)
(using out-of-bounds x=1000)
- Read returns secret byte = **09**
(in cache ⇒ fast)

Memory & Cache Status

array1_size = 00000008

Memory at array1 base:

8 bytes of data (value doesn't matter)

Memory at array1 base+1000:

09 F1 98 CC 90... (something secret)

array2[0*4096]
array2[1*4096]
array2[2*4096]
array2[3*4096]
array2[4*4096]
array2[5*4096]
array2[6*4096]
array2[7*4096]
array2[8*4096]
array2[9*4096]
array2[10*4096]
array2[11*4096]

..

Contents don't matter
only care about cache **status**

Uncached

Cached

Spectre variant 1 attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Next:

- Request mem at (array2 base + 09*4096)
- Brings array2 [09*4096] into the cache
- Realize if() is false: discard speculative work

Finish operation & return to caller

Memory & Cache Status

array1_size = 00000008

Memory at array1 base:

8 bytes of data (value doesn't matter)

Memory at array1 base+1000:

09 F1 98 CC 90... (something secret)

array2[0*4096]
array2[1*4096]
array2[2*4096]
array2[3*4096]
array2[4*4096]
array2[5*4096]
array2[6*4096]
array2[7*4096]
array2[8*4096]
array2[9*4096]
array2[10*4096]
array2[11*4096]

..

Contents don't matter
only care about cache **status**

Uncached

Cached

Spectre variant 1 attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Attacker:

- measures read time for `array2[i*4096]`
- Read for `i=09` is fast (cached),
reveals secret byte !!
- Repeat with many x (10KB/s)

Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base:

8 bytes of data (value doesn't matter)

Memory at `array1` base+1000:

09 F1 98 CC 90... (something secret)

array2[0*4096]
array2[1*4096]
array2[2*4096]
array2[3*4096]
array2[4*4096]
array2[5*4096]
array2[6*4096]
array2[7*4096]
array2[8*4096]
array2[9*4096]
array2[10*4096]
array2[11*4096]

..

Contents don't matter
only care about cache **status**

Uncached

Cached

Fault Attacks

"Using Memory Errors to Attack a Virtual Machine" Govindavajhala Appel 2003

Java heap overflow via glitched address of function pointer.

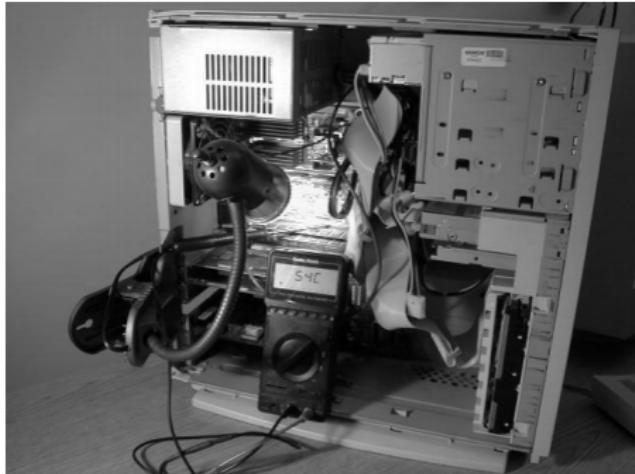


Figure 3. Experimental setup to induce memory errors, showing a PC built from surplus components, clip-on gooseneck lamp, 50-watt spotlight bulb, and digital thermometer. Not shown is the variable AC power supply for the lamp.

Types of RAM

- Volatile memory: Data retained only as long as power is on
- Persistent memory like flash or magnetic disks retains data without power

SRAM

- SRAM retains bit value as long as power is on without any refresh
- Faster, lower density, higher cost
- Has a “burn-in” phenomenon where on startup it tends to flip bit to “remembered bit”

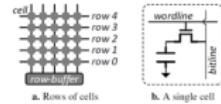
DRAM

- DRAM requires periodic refresh to retain stored value
- Capacitors charged to store data
- Higher density, lowered cost
- “Cold boot attacks” exploited capacitor discharge time to read sensitive data from physical memory

Rowhammer attacks

Seaborn and Dullien 2015

- DRAM cells are grouped into rows
 - All cells in a row are refreshed together
-
- Repeatedly opening and closing a row within a refresh interval causes disturbance errors in adjacent rows.
 - Attacker running attack process on same machine as victim can cause bits to flip in victim's memory



Mitigating timing side channels

- Eliminating all side-channels is practically impossible
- We can eliminate or mitigate **some** channels
 - How do we choose?
 - Attacker model + impact of defense + cost of defense

Sweet spot: timing channels

- Good for the attacker:
 - Remote attackers can exploit timing channels
 - Co-located attacker (on same physical machine) can abuse cache to amplify these attacks
- Good for defense
 - Can eliminate timing channels
 - Performance overhead of doing so is reasonable

To understand how to eliminate the channels
we need to understand what introduces time
variability

Which runs faster?

```
void foo(double x) {  
    double z, y = 1.0;  
    for (uint32_t i = 0; i < 100000000; i++) {  
        z = y*x;  
    }  
}
```

- A: foo(1.0);
- B: foo(1.0e-323);
- C: They take the same amount of time!

Which runs faster?

```
void foo(double x) {  
    double z, y = 1.0;  
    for (uint32_t i = 0; i < 100000000; i++) {  
        z = y*x;  
    }  
}
```

- A: foo(1.0); ←
- B: foo(1.0e-323);
- C: They take the same amount of time!

Why? Floating-point time variability

Processor	+ subnormal	+ special	× subnormal	× special	÷ subnormal	÷ special	÷ x^2	÷ x^4	√subnormal	√special	√ x^2	√ x^4	√ $-x$
	<i>Single-precision operations</i>												
Intel Core i7-7700 (Kaby Lake)	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
Intel Core i7-6700K (Skylake)	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
Intel Core i7-3667U (Ivy Bridge)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗
Intel Xeon X5660 (Westmere)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗
Intel Atom D2550 (Cedarview)	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗
AMD Phenom II X6 1100T	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
AMD Ryzen 7 1800x	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗
<i>Double-precision operations</i>													
Intel Core i7-7700 (Kaby Lake)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Core i7-6700K (Skylake)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Intel Core i7-3667U (Ivy Bridge)	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗
Intel Xeon X5660 (Westmere)	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗
Intel Atom D2550 (Cedarview)	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗
AMD Phenom II X6 1100T	✗	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗
AMD Ryzen 7 1800x	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗

Some instructions introduce time variability

- **Problem:** Certain instructions take different amounts of time depending on the operands
 - If input data is secret: might leak some of it!
- **Solution?**
 - In general, don't use variable-time instructions

When ARMv8.4-DIT is implemented:

Data Independent Timing.

DIT	Meaning
0b0	<p>The architecture makes no statement about the timing properties of any instructions.</p> <p>The architecture requires that:</p> <ul style="list-style-type: none">• The timing of every load and store instruction is insensitive to the value of the data being loaded or stored.• For certain data processing instructions, the instruction takes a time which is independent of:<ul style="list-style-type: none">◦ The values of the data supplied in any of its registers.◦ The values of the NZCV flags.
0b1	<p>◦ The values of the NZCV flags.</p> <ul style="list-style-type: none">• For certain data processing instructions, the response of the instruction to asynchronous exceptions does not vary based on:<ul style="list-style-type: none">◦ The values of the data supplied in any of its registers.◦ The values of the NZCV flags.

The data processing instructions affected by this bit are:

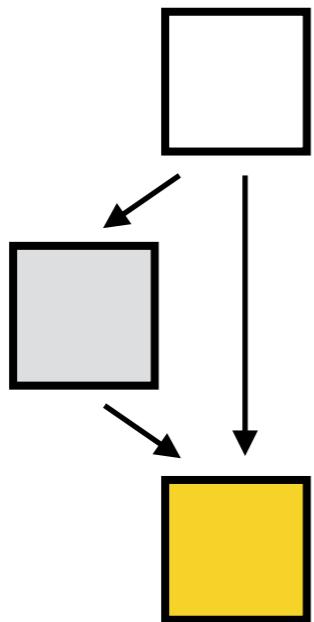
- All cryptographic instructions. These instructions are:
 - AESD, AESE, AESIMC, AESMC, SHA1C, SHA1H, SHA1M, SHA1P, SHA1SU0, SHA1SU1, SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
- A subset of those instructions which use the general-purpose register file. For these instructions, the effects of CPSR.DIT apply only if they do not use R15 as either their source or destination and pass their condition execution check. The instructions are:
 - BFI, BFC, CLZ, CMN, CMP, MLA, MLAS, MLS, MOVT, MUL, MULS, NOP, PKHBT, PKHTB, RBIT, REV, REV16, REVSH, RRX, SADD16, SADD8, SASX, SBFX, SHADD16, SHADD8, SHASX, SHSAX, SHSUB16, SHSUB8, SMLAL**, SMLAW*, SMLSD*, SMMLA*, SMMLS*, SMMUL*, SMUAD*, SMUL*, SSAX, SSUB16, SSUB8, SXTAB*, SXTAH, SXTB*, SXTH, TEQ, TST, UADD*, UASX, UBFX, UHADD*, UHASX, UHSAX, UHSUB*, UMAAL, UMLAL, UMLALS, UMULL, UMULLS, USADA8, USAX, USUB*, UXTAB*, UXTAH, UXTB*, UXTH, ADC (register-shifted register), ADCS (register-shifted register), ADD (register-shifted register), ADDS (register-shifted register), AND (register-shifted register), ANDS (register-shifted register), ASR (register-shifted register), ASRS (register-shifted register), BIC (register-shifted register), BICS (register-shifted register), EOR (register-shifted register), EORS (register-shifted register), LSL (register-shifted register), LSLS (register-shifted register), LSR (register-shifted register), LSRS (register-shifted register), MOV (register-shifted register), MOVS (register-shifted register), MVN (register-shifted register), MVNS (register-shifted register), ORR (register-shifted register), ORRS (register-shifted register), ROR (register-shifted register), RORS (register-shifted register), RSB (register-shifted register), RSBS (register-shifted register), RSC (register-shifted register), RSCS (register-shifted register), SBC (register-shifted register), SBCS (register-shifted register), SUB (register-shifted register), and SUBS (register-shifted register).

Control flow introduces time variability

```
m=1
for i = 0 ... len(d):
    if d[i] = 1: ←
        m = c * m mod N
        m = square(m) mod N
return m
```

if-statements on secrets are unsafe

```
s0;  
if (secret) {  
    s1;  
    s2;  
}  
s3;
```



secret	run	
true	s0;s1;s2;s3;	4 ⌚
false	s0;s3;	2

Can we pad else branch?

```
if (secret) {  
    s1;  
    s2;  
} else {  
    s1';  
    s2';  
}
```

where s1 and s1' take
same amount of time

Why padding branches doesn't work

- **Problem:** Instructions are loaded from cache
 - Which instructions were loaded (or not) observable
- **Problem:** Hardware tried to predict where branch goes
 - Success (or failure) of prediction is observable
- **What can we do?**

Don't branch on secrets!

Real code needs to branch...

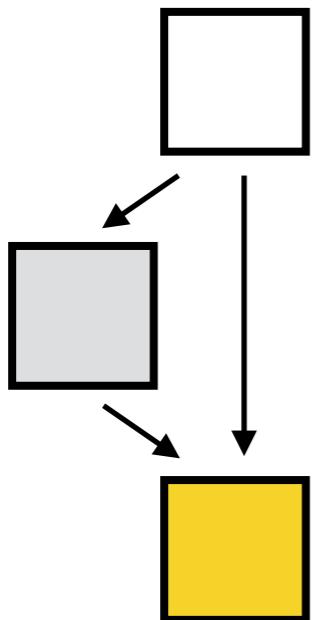
Fold control flow into data flow

(assumption secret = 1 or 0)

```
if (secret) {  
    x = a;  
}
```



```
x = secret * a  
+ (1-secret) * x;
```



Fold control flow into data flow

(assumption secret = 1 or 0)

```
if (secret) {  
    x = a;  
} else {  
    x = b;  
}
```



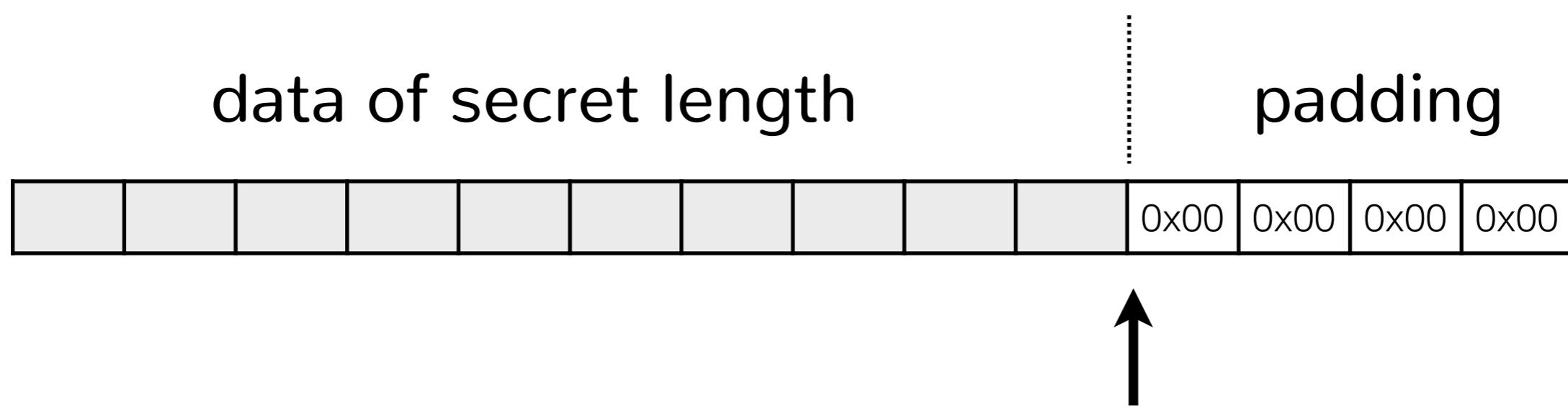
```
x = secret * a  
+ (1-secret) * x;  
  
x = (1-secret) * b  
+ secret * x;
```

Fold control flow into data flow

- Multiple ways to fold control flow into data flow
 - Previous example: takes advantage of arithmetic
 - What's another way?

```
/* Constant-time helper macro that selects l or r depending on all-1 or all-0
 * mask m */
#define CT_SEL(m, l, r) (((m) & (l)) | (~(m) & (r)))
```

An example from mbedTLS



Goal: get the length of the padding so we can remove it

An example from mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return(MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA);

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if( input[i-1] != 0) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

An example from mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if( input[i-1] != 0 ) {
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

Is this safe?

An example from mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i;

    if( NULL == input || NULL == data_len )
        return(MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA);

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        if( input[i-1] != 0) { ←
            *data_len = i;
            return 0;
        }
    }

    return 0;
}
```

Is this safe?

An example from mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        if( done & !prev_done) {
            *data_len = i;
        }
    }

    return 0;
}
```

Is this safe?

An example from mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return( MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA );

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        if( done & !prev_done) { ←
            *data_len = i;
        }
    }

    return 0;
}
```

Is this safe?

An example from mbedTLS

```
static int get_zeros_padding( unsigned char *input, size_t input_len,
                             size_t *data_len )
{
    size_t i
    unsigned done = 0, prev_done = 0;

    if( NULL == input || NULL == data_len )
        return(MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA);

    *data_len = 0;
    for( i = input_len; i > 0; i-- ) {
        prev_done = done;
        done |= input[i-1] != 0;
        *data_len = CT_SEL(done & !prev_done, i, *data_len);
    }

    return 0;
}
```

Is this safe?

Control flow introduces time variability

- **Problem:** Control flow that depends on secret data can lead to information leakage
 - Loops
 - If-statements (switch, etc.)
 - Early returns, goto, break, continue
 - Function calls
- **Solution:** control flow should not depend on secrets, fold secret control flow into data!

Memory access patterns introduce time variability

```
static void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key) {  
...  
// All other round keys are found from the previous round keys.  
for (i = Nk; i < Nb * (Nr + 1); ++i)  
{  
...  
    k = (i - 1) * 4;  
    tempa[0]=RoundKey[k + 0];  
    tempa[1]=RoundKey[k + 1];  
    tempa[2]=RoundKey[k + 2];  
    tempa[3]=RoundKey[k + 3];  
...  
    tempa[0] = sbox[tempa[0]];  
    tempa[1] = sbox[tempa[1]];  
    tempa[2] = sbox[tempa[2]];  
    tempa[3] = sbox[tempa[3]];  
...  
}
```

How do we fix this?

- Only access memory at public index
- How do we express arr[secret]?

x=arr[secret] → `for(size_t i = 0; i < arr_len; i++)
 x = CT_SEL(EQ(secret, i), arr[i], x)`

```
/* Constant-time helper macro that selects l or r depending on all-1 or all-0  
 * mask m */  
#define CT_SEL(m, l, r) (((m) & (l)) | (~(m) & (r)))
```

Summary: what introduces time variability?

- Duration of certain operations depends on data
 - Do not use operators that are variable time
- Control flow
 - Do not branch based on a secret
- Memory access
 - Do not access memory based on a secret

Solution: constant-time programming

- Duration of certain operations depends on data
 - Transform to safe, known CT operations
- Control flow
 - Turn control flow into data flow problem: select!
- Memory access
 - Loop over public bounds of array!

Writing CT code is unholy

```
380 383     SSL3_RECORD *rr;
384 384     unsigned int mac_size;
385 385     unsigned char md[EVP_MAX_MD_SIZE];
386 386 +    int decryption_failed_or_bad_record_mac = 0;
387 387
388 388     rr = &(s->s3->rrec);
389 389
390 390     00 -417,13 +418,10 @0 dtls1_process_record(SSL *s)
391 418         enc_err = s->method->ss13_enc->enc(s,0);
392 419         if (enc_err <= 0)
393 420             {
394 420                 /* decryption failed, silently discard message */
395 421                 if (enc_err < 0)
396 422                     {
397 423                         rr->length = 0;
398 424                         s->packet_length = 0;
399 425                     }
400 426
401 427                 goto err;
402 428
403 429                 /* To minimize information leaked via timing, we will always
404 430                  * perform all computations before discarding the message.
405 431
406 432                 */
407 433                 decryption_failed_or_bad_record_mac = 1;
408 434             }
409 435
410 436     #ifdef TLS_DEBUG
411 437         00 -453,7 +451,7 @0 printf("\n");
412 451             SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_PRE_MAC_LENGTH_TOO_LONG);
413 452             goto f_err;
414 453         #else
415 454             goto err;
416 455         #endif
417 456
418 457             /* check the MAC for rr->input (it's in mac_size bytes at the tail) */
419 458         00 -464,17 +462,25 @0 printf("\n");
420 462             SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_LENGTH_TOO_SHORT);
421 463             goto f_err;
422 464         #else
423 465             goto err;
424 466         #endif
425 467
426 468             rr->length+=mac_size;
427 469             ias->method->ss13_enc->mac(s,md,0);
428 470             if (1 < 0 || memcmp(md,&(rr->data[rr->length]),mac_size) != 0)
429 471                 {
430 472                     goto err;
431 473                     decryption_failed_or_bad_record_mac = 1;
432 474
433 475                 }
434 476
435 477             if (decryption_failed_or_bad_record_mac)
436 478                 {
437 479                     /* decryption failed, silently discard message */
438 480                     rr->length = 0;
439 481                     s->packet_length = 0;
440 482                     goto err;
441 483
442 484             /* r->length is now just compressed */
443 485             if (s->expand != NULL)
444 486                 {
```

OpenSSL padding oracle attack

Canvel, et al. "Password Interception in a SSL/TLS Channel." Crypto, Vol. 2729. 2003.

Writing CT code is unholy

```
380 383     SSL3_RECORD *rr;
384 384     unsigned int mac_size;
385 385     unsigned char md[EVP_MAX_MD_SIZE];
386 386 +     int decryption_failed_or_bad_record_mac = 0;
387 387
388 388     EVP_DigestUpdate(&md_ctx,md,2);
389 389     EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
390 390     EVP_DigestFinal_ex( &md_ctx,md,NULL);
391 391
392 392     EVP_MD_CTX_copy_ex( &md_ctx,hash);
393 393     EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
394 394     EVP_DigestUpdate(&md_ctx,ss13_pad_2,npad);
395 395     EVP_DigestUpdate(&md_ctx,md,md_size);
396 396     EVP_DigestFinal_ex( &md_ctx,md,&md_size);
397 397
398 398     EVP_MD_CTX_cleanup(&md_ctx);
399 399     if (!isend &&
400 400         EVP_CIPHER_CTX_mode(ssl->enc_read_ctx) == EVP_CIPH_CBC_MODE &&
401 401         ss13_cbc_record_digest_supported(hash))
402 402     {
403 403         /* This is a CBC-encrypted record. We must avoid leaking any
404 404         * timing-side channel information about how many blocks of
405 405         * data we are hashing because that gives an attacker a
406 406         * timing-oracle. */
407 407
408 408         /* npad is, at most, 48 bytes and that's with MD5:
409 409         *   16 + 48 + 8 (sequence bytes) + 1 + 2 = 75.
410 410         *
411 411         * With SHA-1 (the largest hash speced for SSLv3) the hash size
412 412         * goes up 4, but npad goes down by 8, resulting in a smaller
413 413         * total size. */
414 414         unsigned char header[75];
415 415         unsigned j = 0;
416 416         memcpy(header+j, mac_sec, md_size);
417 417         j += md_size;
418 418         memcpy(header+j, ss13_pad_1, npad);
419 419         j += npad;
420 420         memcpy(header+j, seq, 8);
421 421         j += 8;
422 422         header[j++] = rec->type;
423 423         header[j++] = rec->length >> 8;
424 424         header[j++] = rec->length & 0xff;
425 425
426 426         ss13_cbc_digest_record(
427 427             hash,
428 428             md, &md_size,
429 429             header, rec->input,
430 430             rec->length + md_size, rec->orig_len,
431 431             mac_sec, md_size,
432 432             1 /* is SSLv3 */);
433 433     }
434 434     else
435 435     {
436 436         unsigned int md_size_u;
437 437         /* Chop the digest off the end :-)
438 438         EVP_MD_CTX_init(&md_ctx);
439 439
440 440         EVP_MD_CTX_copy_ex( &md_ctx,hash);
441 441         EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
442 442         EVP_DigestUpdate(&md_ctx,ss13_pad_1,npad);
443 443         EVP_DigestUpdate(&md_ctx,seq,8);
444 444         rec_char=rec->type;
445 445         EVP_DigestUpdate(&md_ctx,&rec_char,1);
446 446         p=&md;
447 447         s2n(rec->length,p);
448 448         EVP_DigestUpdate(&md_ctx,md,2);
449 449         EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
450 450
451 451
452 452
453 453
454 454
455 455
456 456
457 457
458 458
459 459
460 460
461 461
462 462
463 463
464 464
465 465
466 466
467 467
468 468
469 469
470 470
471 471
472 472
473 473
474 474
475 475
476 476
477 477
478 478
479 479
480 480
481 481
482 482
483 483
484 484
485 485
486 486
487 487
488 488
489 489
490 490
491 491
492 492
493 493
494 494
495 495
496 496
497 497
498 498
499 499
500 500
501 501
502 502
503 503
504 504
505 505
506 506
507 507
508 508
509 509
510 510
511 511
512 512
513 513
514 514
515 515
516 516
517 517
518 518
519 519
520 520
521 521
522 522
523 523
524 524
525 525
526 526
527 527
528 528
529 529
530 530
531 531
532 532
533 533
534 534
535 535
536 536
537 537
538 538
539 539
540 540
541 541
542 542
543 543
544 544
545 545
546 546
547 547
548 548
549 549
550 550
551 551
552 552
553 553
554 554
555 555
556 556
557 557
558 558
559 559
560 560
561 561
562 562
563 563
564 564
565 565
566 566
567 567
568 568
569 569
570 570
571 571
572 572
573 573
574 574
575 575
576 576
577 577
578 578
579 579
580 580
581 581
582 582
583 583
584 584
585 585
586 586
587 587
588 588
589 589
590 590
591 591
592 592
593 593
594 594
595 595
596 596
597 597
598 598
599 599
600 600
601 601
602 602
603 603
604 604
605 605
606 606
607 607
608 608
609 609
610 610
611 611
612 612
613 613
614 614
615 615
616 616
617 617
618 618
619 619
620 620
621 621
622 622
623 623
624 624
625 625
626 626
627 627
628 628
629 629
630 630
631 631
632 632
633 633
634 634
635 635
636 636
637 637
638 638
639 639
640 640
641 641
642 642
643 643
644 644
645 645
646 646
647 647
648 648
649 649
650 650
651 651
652 652
653 653
654 654
655 655
656 656
657 657
658 658
659 659
660 660
661 661
662 662
663 663
664 664
665 665
666 666
667 667
668 668
669 669
670 670
671 671
672 672
673 673
674 674
675 675
676 676
677 677
678 678
679 679
680 680
681 681
682 682
683 683
684 684
685 685
686 686
687 687
688 688
689 689
690 690
691 691
692 692
693 693
694 694
695 695
696 696
697 697
698 698
699 699
700 700
701 701
702 702
703 703
704 704
705 705
706 706
707 707
708 708
709 709
710 710
711 711
712 712
713 713
714 714
715 715
716 716
717 717
718 718
719 719
720 720
721 721
722 722
723 723
724 724
725 725
726 726
727 727
728 728
729 729
730 730
731 731
732 732
733 733
734 734
735 735
736 736
737 737
738 738
739 739
740 740
741 741
742 742
743 743
744 744
745 745
746 746
747 747
748 748
749 749
750 750
751 751
752 752
753 753
754 754
755 755
756 756
757 757
758 758
759 759
760 760
761 761
762 762
763 763
764 764
765 765
766 766
767 767
768 768
769 769
770 770
771 771
772 772
773 773
774 774
775 775
776 776
777 777
778 778
779 779
780 780
781 781
782 782
783 783
784 784
785 785
786 786
787 787
788 788
789 789
790 790
791 791
792 792
793 793
794 794
795 795
796 796
797 797
798 798
799 799
800 800
801 801
802 802
803 803
804 804
805 805
806 806
807 807
808 808
809 809
810 810
811 811
812 812
813 813
814 814
815 815
816 816
817 817
818 818
819 819
820 820
821 821
822 822
823 823
824 824
825 825
826 826
827 827
828 828
829 829
830 830
831 831
832 832
833 833
834 834
835 835
836 836
837 837
838 838
839 839
840 840
841 841
842 842
843 843
844 844
845 845
846 846
847 847
848 848
849 849
850 850
851 851
852 852
853 853
854 854
855 855
856 856
857 857
858 858
859 859
860 860
861 861
862 862
863 863
864 864
865 865
866 866
867 867
868 868
869 869
870 870
871 871
872 872
873 873
874 874
875 875
876 876
877 877
878 878
879 879
880 880
881 881
882 882
883 883
884 884
885 885
886 886
887 887
888 888
889 889
890 890
891 891
892 892
893 893
894 894
895 895
896 896
897 897
898 898
899 899
900 900
901 901
902 902
903 903
904 904
905 905
906 906
907 907
908 908
909 909
910 910
911 911
912 912
913 913
914 914
915 915
916 916
917 917
918 918
919 919
920 920
921 921
922 922
923 923
924 924
925 925
926 926
927 927
928 928
929 929
930 930
931 931
932 932
933 933
934 934
935 935
936 936
937 937
938 938
939 939
940 940
941 941
942 942
943 943
944 944
945 945
946 946
947 947
948 948
949 949
950 950
951 951
952 952
953 953
954 954
955 955
956 956
957 957
958 958
959 959
960 960
961 961
962 962
963 963
964 964
965 965
966 966
967 967
968 968
969 969
970 970
971 971
972 972
973 973
974 974
975 975
976 976
977 977
978 978
979 979
980 980
981 981
982 982
983 983
984 984
985 985
986 986
987 987
988 988
989 989
990 990
991 991
992 992
993 993
994 994
995 995
996 996
997 997
998 998
999 999
1000 1000
```

OpenSSL padding oracle attack
Canvel, et al. “Password Interception in a
SSL/TLS Channel.” Crypto, Vol. 2729. 2003.

Writing CT code is unholy

```
380 383     SSL3_RECORD *rr;
384 384     unsigned int mac_size;
385 385     unsigned char md[EVP_MAX_MD_SIZE];
386 386 +     int decryption_failed_or_bad_record_mac = 0;
387 387
388 388     EVP_DigestUpdate(&md_ctx,md,2);
389 389     EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
390 390     EVP_DigestFinal_ex( &md_ctx,md,NULL);
391 391
392 392     EVP_MD_CTX_copy_ex( &md_ctx,hash);
393 393     EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
394 394     EVP_DigestUpdate(&md_ctx,ss13_pad_2,npad);
395 395     EVP_DigestUpdate(&md_ctx,md,md_size);
396 396     EVP_DigestFinal_ex( &md_ctx,md,&md_size);
397 397
398 398     EVP_MD_CTX_cleanup(&md_ctx);
399 399     if (!isend &&
400 400         EVP_CIPHER_CTX_mode(ssl->enc_read_ctx) == EVP_CIPH_CBC_MODE &&
401 401         ss13_cbc_record_digest_supported(hash))
402 402     {
403 403         /* This is a CBC-encrypted record. We must avoid leaking any
404 404         * timing-side channel information about how many blocks of
405 405         * data we are hashing because that gives an attacker a
406 406         * timing-oracle. */
407 407
408 408         /* npad is, at most, 48 bytes and that's with MD5:
409 409         *   16 + 48 + 8 (sequence bytes) + 1 + 2 = 75.
410 410         *
411 411         * With SHA-1 (the largest hash speced for SSLv3) the hash size
412 412         * goes up 4, but npad goes down by 8, resulting in a smaller
413 413         * total size. */
414 414         unsigned char header[75];
415 415         unsigned j = 0;
416 416         memcpy(header+j, mac_sec, md_size);
417 417         j += md_size;
418 418         memcpy(header+j, ss13_pad_1, npad);
419 419         j += npad;
420 420         memcpy(header+j, seq, 8);
421 421         j += 8;
422 422         header[j++] = rec->type;
423 423         header[j++] = rec->length >> 8;
424 424         header[j++] = rec->length & 0xff;
425 425
426 426         ss13_cbc_digest_record(
427 427             hash,
428 428             md, &md_size,
429 429             header, rec->input,
430 430             rec->length + md_size, rec->orig_len,
431 431             mac_sec, md_size,
432 432             1 /* is SSLv3 */);
433 433     }
434 434     else
435 435     {
436 436         unsigned int md_size_u;
437 437         /* Chop the digest off the end :-)
438 438         EVP_MD_CTX_init(&md_ctx);
439 439
440 440         EVP_MD_CTX_copy_ex( &md_ctx,hash);
441 441         EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
442 442         EVP_DigestUpdate(&md_ctx,ss13_pad_1,npad);
443 443         EVP_DigestUpdate(&md_ctx,seq,8);
444 444         rec_char=rec->type;
445 445         EVP_DigestUpdate(&md_ctx,&rec_char,1);
446 446         p=&md;
447 447         s2n(rec->length,p);
448 448         EVP_DigestUpdate(&md_ctx,md,2);
449 449         EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
450 450
451 451
452 452
453 453
454 454
455 455
456 456
457 457
458 458
459 459
460 460
461 461
462 462
463 463
464 464
465 465
466 466
467 467
468 468
469 469
470 470
471 471
472 472
473 473
474 474
475 475
476 476
477 477
478 478
479 479
480 480
481 481
482 482
483 483
484 484
485 485
486 486
487 487
488 488
489 489
490 490
491 491
492 492
493 493
494 494
495 495
496 496
497 497
498 498
499 499
500 500
501 501
502 502
503 503
504 504
505 505
506 506
507 507
508 508
509 509
510 510
511 511
512 512
513 513
514 514
515 515
516 516
517 517
518 518
519 519
520 520
521 521
522 522
523 523
524 524
525 525
526 526
527 527
528 528
529 529
530 530
531 531
532 532
533 533
534 534
535 535
536 536
537 537
538 538
539 539
540 540
541 541
542 542
543 543
544 544
545 545
546 546
547 547
548 548
549 549
550 550
551 551
552 552
553 553
554 554
555 555
556 556
557 557
558 558
559 559
560 560
561 561
562 562
563 563
564 564
565 565
566 566
567 567
568 568
569 569
570 570
571 571
572 572
573 573
574 574
575 575
576 576
577 577
578 578
579 579
580 580
581 581
582 582
583 583
584 584
585 585
586 586
587 587
588 588
589 589
590 590
591 591
592 592
593 593
594 594
595 595
596 596
597 597
598 598
599 599
600 600
601 601
602 602
603 603
604 604
605 605
606 606
607 607
608 608
609 609
610 610
611 611
612 612
613 613
614 614
615 615
616 616
617 617
618 618
619 619
620 620
621 621
622 622
623 623
624 624
625 625
626 626
627 627
628 628
629 629
630 630
631 631
632 632
633 633
634 634
635 635
636 636
637 637
638 638
639 639
640 640
641 641
642 642
643 643
644 644
645 645
646 646
647 647
648 648
649 649
650 650
651 651
652 652
653 653
654 654
655 655
656 656
657 657
658 658
659 659
660 660
661 661
662 662
663 663
664 664
665 665
666 666
667 667
668 668
669 669
670 670
671 671
672 672
673 673
674 674
675 675
676 676
677 677
678 678
679 679
680 680
681 681
682 682
683 683
684 684
685 685
686 686
687 687
688 688
689 689
690 690
691 691
692 692
693 693
694 694
695 695
696 696
697 697
698 698
699 699
700 700
701 701
702 702
703 703
704 704
705 705
706 706
707 707
708 708
709 709
710 710
711 711
712 712
713 713
714 714
715 715
716 716
717 717
718 718
719 719
720 720
721 721
722 722
723 723
724 724
725 725
726 726
727 727
728 728
729 729
730 730
731 731
732 732
733 733
734 734
735 735
736 736
737 737
738 738
739 739
740 740
741 741
742 742
743 743
744 744
745 745
746 746
747 747
748 748
749 749
750 750
751 751
752 752
753 753
754 754
755 755
756 756
757 757
758 758
759 759
760 760
761 761
762 762
763 763
764 764
765 765
766 766
767 767
768 768
769 769
770 770
771 771
772 772
773 773
774 774
775 775
776 776
777 777
778 778
779 779
780 780
781 781
782 782
783 783
784 784
785 785
786 786
787 787
788 788
789 789
790 790
791 791
792 792
793 793
794 794
795 795
796 796
797 797
798 798
799 799
800 800
801 801
802 802
803 803
804 804
805 805
806 806
807 807
808 808
809 809
810 810
811 811
812 812
813 813
814 814
815 815
816 816
817 817
818 818
819 819
820 820
821 821
822 822
823 823
824 824
825 825
826 826
827 827
828 828
829 829
830 830
831 831
832 832
833 833
834 834
835 835
836 836
837 837
838 838
839 839
840 840
841 841
842 842
843 843
844 844
845 845
846 846
847 847
848 848
849 849
850 850
851 851
852 852
853 853
854 854
855 855
856 856
857 857
858 858
859 859
860 860
861 861
862 862
863 863
864 864
865 865
866 866
867 867
868 868
869 869
870 870
871 871
872 872
873 873
874 874
875 875
876 876
877 877
878 878
879 879
880 880
881 881
882 882
883 883
884 884
885 885
886 886
887 887
888 888
889 889
890 890
891 891
892 892
893 893
894 894
895 895
896 896
897 897
898 898
899 899
900 900
901 901
902 902
903 903
904 904
905 905
906 906
907 907
908 908
909 909
910 910
911 911
912 912
913 913
914 914
915 915
916 916
917 917
918 918
919 919
920 920
921 921
922 922
923 923
924 924
925 925
926 926
927 927
928 928
929 929
930 930
931 931
932 932
933 933
934 934
935 935
936 936
937 937
938 938
939 939
940 940
941 941
942 942
943 943
944 944
945 945
946 946
947 947
948 948
949 949
950 950
951 951
952 952
953 953
954 954
955 955
956 956
957 957
958 958
959 959
960 960
961 961
962 962
963 963
964 964
965 965
966 966
967 967
968 968
969 969
970 970
971 971
972 972
973 973
974 974
975 975
976 976
977 977
978 978
979 979
980 980
981 981
982 982
983 983
984 984
985 985
986 986
987 987
988 988
989 989
990 990
991 991
992 992
993 993
994 994
995 995
996 996
997 997
998 998
999 999
1000 1000
```

OpenSSL padding oracle attack
Canvel, et al. "Password Interception in a
SSL/TLS Channel." Crypto, Vol. 2729. 2003.

Lucky 13 timing attack
Al Fardan and Paterson. "Lucky thirteen:
Breaking the TLS and DTLS record
protocols." Oakland 2013.

Writing CT code is unholy



```
380 383     SSL3_RECORD *rr;
384 384     unsigned int mac_size;
385 385     unsigned char md[EVP_MAX_MD_SIZE];
386 386 +     int decryption_failed_or_bad_record_mac = 0;
387 387
388 388     EVP_DigestUpdate(&md_ctx,md,2);
389 389     EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
390 390     EVP_DigestFinal_ex( &md_ctx,md,NULL);
391 391
392 392     unsigned char mac[SHA_DIGEST_LENGTH];
393 393     union { unsigned int u[SHA_DIGEST_LENGTH/sizeof(unsigned int)];
394 394         unsigned char c[SHA_DIGEST_LENGTH]; } mac;
395 395
396 396     /* decrypt HMAC(padding at once */
397 397     aesni_cbc_encrypt(in,out,len,
398 398         &key->ks,ctx->iv,0);
399 399
400 400     if (plen) { /* "TLS" mode of operation */
401 401         /* figure out payload length */
402 402         if (len<(size_t)(out[len-1]+1+SHA_DIGEST_LENGTH))
403 403             return 0;
404 404
405 405         len -= (out[len-1]+1+SHA_DIGEST_LENGTH);
406 406         size_t inp_len, mask, j, 1;
407 407         unsigned int res, maxpad, pad, bitlen;
408 408         int ret = 1;
409 409         union { unsigned int u[SHA_CBLOCK];
410 410             unsigned char c[SHA_CBLOCK]; }
411 411         *data = (void *)key->md.data;
412 412
413 413         if ((key->aux.tls_aad[plen-4]<=8||key->aux.tls_aad[plen-3])
414 414             >= TLS1_1_VERSION) {
415 415             len -= AES_BLOCK_SIZE;
416 416             >= TLS1_1_VERSION
417 417             iv = AES_BLOCK_SIZE;
418 418         }
419 419
420 420         key->aux.tls_aad[plen-2] = len>>8;
421 421         key->aux.tls_aad[plen-1] = len;
422 422         if (len<(iv+SHA_DIGEST_LENGTH+1))
423 423             return 0;
424 424
425 425         /* omit explicit iv */
426 426         out += iv;
427 427         len -= iv;
428 428
429 429         /* figure out payload length */
430 430         pad = out[len-1];
431 431         maxpad = len-(SHA_DIGEST_LENGTH+1);
432 432         maxpad |= (255-maxpad)>>(sizeof(maxpad)*8-8);
433 433         maxpad &= 255;
434 434
435 435         inp_len = len - (SHA_DIGEST_LENGTH+pad+1);
436 436         mask = (0-((inp_len-len)>>(sizeof(inp_len)*8-1)));
437 437         inp_len &= mask;
438 438         ret &= (int)mask;
439 439
440 440         /* calculate HMAC and verify it */
441 441         key->aux.tls_aad[plen-2] = inp_len>>8;
442 442         key->aux.tls_aad[plen-1] = inp_len;
443 443
444 444         /* calculate HMAC */
445 445         key->md = key->head;
446 446         SHA1_Update(&key->md,key->aux.tls_aad,plen);
447 447         SHA1_Update(&key->md,out+iv,len);
448 448         SHA1_Final(mac,&key->md);
449 449
450 450     #if 1
451 451         len -= SHA_DIGEST_LENGTH; /* amend mac */
452 452         if (len>=(256*SHA_CBLOCK)) {
453 453             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
454 454             j += SHA_CBLOCK-key->md.num;
455 455             SHA1_Update(&key->md,out+j);
456 456
457 457         }
458 458
459 459     }
460 460
461 461     if (decryption_failed_or_bad_record_mac)
462 462         return -1;
463 463
464 464     if (plen)
465 465         return len;
466 466
467 467     if (mac_size)
468 468         return len+mac_size;
469 469
470 470     if (len)
471 471         return len;
472 472
473 473     if (mac_size)
474 474         return len+mac_size;
475 475
476 476     if (len)
477 477         return len;
478 478
479 479     if (mac_size)
480 480         return len+mac_size;
481 481
482 482     if (len)
483 483         return len;
484 484
485 485     if (mac_size)
486 486         return len+mac_size;
487 487
488 488     if (len)
489 489         return len;
490 490
491 491     if (mac_size)
492 492         return len+mac_size;
493 493
494 494     if (len)
495 495         return len;
496 496
497 497     if (mac_size)
498 498         return len+mac_size;
499 499
500 500     if (len)
501 501         return len;
502 502
503 503     if (mac_size)
504 504         return len+mac_size;
505 505
506 506     if (len)
507 507         return len;
508 508
509 509     if (mac_size)
510 510         return len+mac_size;
511 511
512 512     if (len)
513 513         return len;
514 514
515 515     if (mac_size)
516 516         return len+mac_size;
517 517
518 518     if (len)
519 519         return len;
520 520
521 521     if (mac_size)
522 522         return len+mac_size;
523 523
524 524     if (len)
525 525         return len;
526 526
527 527     if (mac_size)
528 528         return len+mac_size;
529 529
530 530     if (len)
531 531         return len;
532 532
533 533     if (mac_size)
534 534         return len+mac_size;
535 535
536 536     if (len)
537 537         return len;
538 538
539 539     if (mac_size)
540 540         return len+mac_size;
541 541
542 542     if (len)
543 543         return len;
544 544
545 545     if (mac_size)
546 546         return len+mac_size;
547 547
548 548     if (len)
549 549         return len;
550 550
551 551     if (mac_size)
552 552         return len+mac_size;
553 553
554 554     if (len)
555 555         return len;
556 556
557 557     if (mac_size)
558 558         return len+mac_size;
559 559
560 560     if (len)
561 561         return len;
562 562
563 563     if (mac_size)
564 564         return len+mac_size;
565 565
566 566     if (len)
567 567         return len;
568 568
569 569     if (mac_size)
570 570         return len+mac_size;
571 571
572 572     if (len)
573 573         return len;
574 574
575 575     if (mac_size)
576 576         return len+mac_size;
577 577
578 578     if (len)
579 579         return len;
580 580
581 581     if (mac_size)
582 582         return len+mac_size;
583 583
584 584     if (len)
585 585         return len;
586 586
587 587     if (mac_size)
588 588         return len+mac_size;
589 589
590 590     if (len)
591 591         return len;
592 592
593 593     if (mac_size)
594 594         return len+mac_size;
595 595
596 596     if (len)
597 597         return len;
598 598
599 599     if (mac_size)
600 600         return len+mac_size;
601 601
602 602     if (len)
603 603         return len;
604 604
605 605     if (mac_size)
606 606         return len+mac_size;
607 607
608 608     if (len)
609 609         return len;
610 610
611 611     if (mac_size)
612 612         return len+mac_size;
613 613
614 614     if (len)
615 615         return len;
616 616
617 617     if (mac_size)
618 618         return len+mac_size;
619 619
620 620     if (len)
621 621         return len;
622 622
623 623     if (mac_size)
624 624         return len+mac_size;
625 625
626 626     if (len)
627 627         return len;
628 628
629 629     if (mac_size)
630 630         return len+mac_size;
631 631
632 632     if (len)
633 633         return len;
634 634
635 635     if (mac_size)
636 636         return len+mac_size;
637 637
638 638     if (len)
639 639         return len;
640 640
641 641     if (mac_size)
642 642         return len+mac_size;
643 643
644 644     if (len)
645 645         return len;
646 646
647 647     if (mac_size)
648 648         return len+mac_size;
649 649
650 650     if (len)
651 651         return len;
652 652
653 653     if (mac_size)
654 654         return len+mac_size;
655 655
656 656     if (len)
657 657         return len;
658 658
659 659     if (mac_size)
660 660         return len+mac_size;
661 661
662 662     if (len)
663 663         return len;
664 664
665 665     if (mac_size)
666 666         return len+mac_size;
667 667
668 668     if (len)
669 669         return len;
670 670
671 671     if (mac_size)
672 672         return len+mac_size;
673 673
674 674     if (len)
675 675         return len;
676 676
677 677     if (mac_size)
678 678         return len+mac_size;
679 679
680 680     if (len)
681 681         return len;
682 682
683 683     if (mac_size)
684 684         return len+mac_size;
685 685
686 686     if (len)
687 687         return len;
688 688
689 689     if (mac_size)
690 690         return len+mac_size;
691 691
692 692     if (len)
693 693         return len;
694 694
695 695     if (mac_size)
696 696         return len+mac_size;
697 697
698 698     if (len)
699 699         return len;
700 700
701 701     if (mac_size)
702 702         return len+mac_size;
703 703
704 704     if (len)
705 705         return len;
706 706
707 707     if (mac_size)
708 708         return len+mac_size;
709 709
710 710     if (len)
711 711         return len;
712 712
713 713     if (mac_size)
714 714         return len+mac_size;
715 715
716 716     if (len)
717 717         return len;
718 718
719 719     if (mac_size)
720 720         return len+mac_size;
721 721
722 722     if (len)
723 723         return len;
724 724
725 725     if (mac_size)
726 726         return len+mac_size;
727 727
728 728     if (len)
729 729         return len;
730 730
731 731     if (mac_size)
732 732         return len+mac_size;
733 733
734 734     if (len)
735 735         return len;
736 736
737 737     if (mac_size)
738 738         return len+mac_size;
739 739
740 740     if (len)
741 741         return len;
742 742
743 743     if (mac_size)
744 744         return len+mac_size;
745 745
746 746     if (len)
747 747         return len;
748 748
749 749     if (mac_size)
750 750         return len+mac_size;
751 751
752 752     if (len)
753 753         return len;
754 754
755 755     if (mac_size)
756 756         return len+mac_size;
757 757
758 758     if (len)
759 759         return len;
760 760
761 761     if (mac_size)
762 762         return len+mac_size;
763 763
764 764     if (len)
765 765         return len;
766 766
767 767     if (mac_size)
768 768         return len+mac_size;
769 769
770 770     if (len)
771 771         return len;
772 772
773 773     if (mac_size)
774 774         return len+mac_size;
775 775
776 776     if (len)
777 777         return len;
778 778
779 779     if (mac_size)
780 780         return len+mac_size;
781 781
782 782     if (len)
783 783         return len;
784 784
785 785     if (mac_size)
786 786         return len+mac_size;
787 787
788 788     if (len)
789 789         return len;
790 790
791 791     if (mac_size)
792 792         return len+mac_size;
793 793
794 794     if (len)
795 795         return len;
796 796
797 797     if (mac_size)
798 798         return len+mac_size;
799 799
800 800     if (len)
801 801         return len;
802 802
803 803     if (mac_size)
804 804         return len+mac_size;
805 805
806 806     if (len)
807 807         return len;
808 808
809 809     if (mac_size)
810 810         return len+mac_size;
811 811
812 812     if (len)
813 813         return len;
814 814
815 815     if (mac_size)
816 816         return len+mac_size;
817 817
818 818     if (len)
819 819         return len;
820 820
821 821     if (mac_size)
822 822         return len+mac_size;
823 823
824 824     if (len)
825 825         return len;
826 826
827 827     if (mac_size)
828 828         return len+mac_size;
829 829
830 830     if (len)
831 831         return len;
832 832
833 833     if (mac_size)
834 834         return len+mac_size;
835 835
836 836     if (len)
837 837         return len;
838 838
839 839     if (mac_size)
840 840         return len+mac_size;
841 841
842 842     if (len)
843 843         return len;
844 844
845 845     if (mac_size)
846 846         return len+mac_size;
847 847
848 848     if (len)
849 849         return len;
850 850
851 851     if (mac_size)
852 852         return len+mac_size;
853 853
854 854     if (len)
855 855         return len;
856 856
857 857     if (mac_size)
858 858         return len+mac_size;
859 859
860 860     if (len)
861 861         return len;
862 862
863 863     if (mac_size)
864 864         return len+mac_size;
865 865
866 866     if (len)
867 867         return len;
868 868
869 869     if (mac_size)
870 870         return len+mac_size;
871 871
872 872     if (len)
873 873         return len;
874 874
875 875     if (mac_size)
876 876         return len+mac_size;
877 877
878 878     if (len)
879 879         return len;
880 880
881 881     if (mac_size)
882 882         return len+mac_size;
883 883
884 884     if (len)
885 885         return len;
886 886
887 887     if (mac_size)
888 888         return len+mac_size;
889 889
890 890     if (len)
891 891         return len;
892 892
893 893     if (mac_size)
894 894         return len+mac_size;
895 895
896 896     if (len)
897 897         return len;
898 898
899 899     if (mac_size)
900 900         return len+mac_size;
901 901
902 902     if (len)
903 903         return len;
904 904
905 905     if (mac_size)
906 906         return len+mac_size;
907 907
908 908     if (len)
909 909         return len;
910 910
911 911     if (mac_size)
912 912         return len+mac_size;
913 913
914 914     if (len)
915 915         return len;
916 916
917 917     if (mac_size)
918 918         return len+mac_size;
919 919
920 920     if (len)
921 921         return len;
922 922
923 923     if (mac_size)
924 924         return len+mac_size;
925 925
926 926     if (len)
927 927         return len;
928 928
929 929     if (mac_size)
930 930         return len+mac_size;
931 931
932 932     if (len)
933 933         return len;
934 934
935 935     if (mac_size)
936 936         return len+mac_size;
937 937
938 938     if (len)
939 939         return len;
940 940
941 941     if (mac_size)
942 942         return len+mac_size;
943 943
944 944     if (len)
945 945         return len;
946 946
947 947     if (mac_size)
948 948         return len+mac_size;
949 949
950 950     if (len)
951 951         return len;
952 952
953 953     if (mac_size)
954 954         return len+mac_size;
955 955
956 956     if (len)
957 957         return len;
958 958
959 959     if (mac_size)
960 960         return len+mac_size;
961 961
962 962     if (len)
963 963         return len;
964 964
965 965     if (mac_size)
966 966         return len+mac_size;
967 967
968 968     if (len)
969 969         return len;
970 970
971 971     if (mac_size)
972 972         return len+mac_size;
973 973
974 974     if (len)
975 975         return len;
976 976
977 977     if (mac_size)
978 978         return len+mac_size;
979 979
980 980     if (len)
981 981         return len;
982 982
983 983     if (mac_size)
984 984         return len+mac_size;
985 985
986 986     if (len)
987 987         return len;
988 988
989 989     if (mac_size)
990 990         return len+mac_size;
991 991
992 992     if (len)
993 993         return len;
994 994
995 995     if (mac_size)
996 996         return len+mac_size;
997 997
998 998     if (len)
999 999         return len;
1000 1000
1001 1001     if (mac_size)
1002 1002         return len+mac_size;
1003 1003
1004 1004     if (len)
1005 1005         return len;
1006 1006
1007 1007     if (mac_size)
1008 1008         return len+mac_size;
1009 1009
1010 1010     if (len)
1011 1011         return len;
1012 1012
1013 1013     if (mac_size)
1014 1014         return len+mac_size;
1015 1015
1016 1016     if (len)
1017 1017         return len;
1018 1018
1019 1019     if (mac_size)
1020 1020         return len+mac_size;
1021 1021
1022 1022     if (len)
1023 1023         return len;
1024 1024
1025 1025     if (mac_size)
1026 1026         return len+mac_size;
1027 1027
1028 1028     if (len)
1029 1029         return len;
1030 1030
1031 1031     if (mac_size)
1032 1032         return len+mac_size;
1033 1033
1034 1034     if (len)
1035 1035         return len;
1036 1036
1037 1037     if (mac_size)
1038 1038         return len+mac_size;
1039 1039
1040 1040     if (len)
1041 1041         return len;
1042 1042
1043 1043     if (mac_size)
1044 1044         return len+mac_size;
1045 1045
1046 1046     if (len)
1047 1047         return len;
1048 1048
1049 1049     if (mac_size)
1050 1050         return len+mac_size;
1051 1051
1052 1052     if (len)
1053 1053         return len;
1054 1054
1055 1055     if (mac_size)
1056 1056         return len+mac_size;
1057 1057
1058 1058     if (len)
1059 1059         return len;
1060 1060
1061 1061     if (mac_size)
1062 1062         return len+mac_size;
1063 1063
1064 1064     if (len)
1065 1065         return len;
1066 1066
1067 1067     if (mac_size)
1068 1068         return len+mac_size;
1069 1069
1070 1070     if (len)
1071 1071         return len;
1072 1072
1073 1073     if (mac_size)
1074 1074         return len+mac_size;
1075 1075
1076 1076     if (len)
1077 1077         return len;
1078 1078
1079 1079     if (mac_size)
1080 1080         return len+mac_size;
1081 1081
1082 1082     if (len)
1083 1083         return len;
1084 1084
1085 1085     if (mac_size)
1086 1086         return len+mac_size;
1087 1087
1088 1088     if (len)
1089 1089         return len;
1
```

Writing CT code is unholy

The screenshot shows a code editor displaying a portion of the OpenSSL source code. The code is color-coded by file: green for C code, red for comments, and blue for preprocessor directives. Several sections of the code are highlighted with different colors (pink, light green, light red) and framed by black and red lines, likely indicating specific regions of interest for analysis. The code itself is a complex cryptographic function, likely related to TLS or SSL processing.

```
380 383     SSL3_RECORD *rr;
384 384     unsigned int mac_size;
385 385     unsigned char md[EVP_MAX_MD_SIZE];
386 386 +     int decryption_failed_or_bad_record_mac = 0;
387 387
388 388     EVP_DigestUpdate(&md_ctx, md, 2);
389 389     EVP_DigestUpdate(&md_ctx, rec->input, rec->length);
390 390     EVP_DigestFinal_ex( &md_ctx, md, NULL);
391 391
392 392     unsigned char mac[SHA_DIGEST_LENGTH];
393 393     union { unsigned int u[SHA_DIGEST_LENGTH/sizeof(unsigned int)];
394 394         unsigned char c[SHA_DIGEST_LENGTH]; } mac;
395 395
396 396     /* decrypt HMAC(padding at once */
397 397     aesni_cbc_encrypt(in,out,len,
398 398         &key->ks,ctx->iv,0);
399 399
400 400     if (plen) { /* "TLS" mode of operation */
401 401         /* figure out payload length */
402 402         if (len<(size_t)(out[len-1]+SHA_DIGEST_LENGTH))
403 403             return 0;
404 404
405 405         len -= (out[len-1]+SHA_DIGEST_LENGTH);
406 406         size_t inp_len, mask, j, 1;
407 407         unsigned int res, maxpad, pad, bitlen;
408 408         int ret = 1;
409 409         union { unsigned int u[SHA_CBLOCK];
410 410             unsigned char c[SHA_CBLOCK]; }
411 411         *data = (void *)key->md.data;
412 412
413 413         if ((key->aux.tls_aad[plen-4]<=8||key->aux.tls_aad[plen-3])
414 414             >= TLS1_1_VERSION) {
415 415             len -= AES_BLOCK_SIZE;
416 416             >= TLS1_1_VERSION
417 417             iv = AES_BLOCK_SIZE;
418 418         }
419 419
420 420         key->aux.tls_aad[plen-2] = len>>8;
421 421         key->aux.tls_aad[plen-1] = len;
422 422         /* calculate HMAC and verify it */
423 423         maxpad |= (255 - maxpad) >> (sizeof(maxpad) * 8 - 8);
424 424         maxpad &= 255;
425 425
426 426         ret &= constant_time_ge(maxpad, pad);
427 427
428 428         inp_len = len - (SHA_DIGEST_LENGTH + pad + 1);
429 429         mask = (0 - ((inp_len - len) >> (sizeof(inp_len) * 8 - 1)));
430 430         inp_len &= mask;
431 431
432 432         mask = (0-((inp_len-len)>>(sizeof(inp_len)*8-1)));
433 433         inp_len &= mask;
434 434         ret &= (int)mask;
435 435
436 436         /* calculate HMAC and verify it */
437 437         key->md = key->head;
438 438         SHA1_Update(&key->md,key->aux.tls_aadplen);
439 439         SHA1_Update(&key->md,out+iv,len);
440 440         SHA1_Final(mac,&key->md);
441 441
442 442         /* amend mac */
443 443         if (len>(256*SHA_CBLOCK)) {
444 444             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
445 445             j += SHA_CBLOCK-key->md.num;
446 446             SHA1_Update(&key->md,out+j);
447 447
448 448         }
449 449
450 450     }
451 451
452 452     if (decryption_failed_or_bad_record_mac)
453 453         return -1;
454 454
455 455     return 1;
456 456
457 457     /* calculate HMAC */
458 458     key->md = key->head;
459 459     SHA1_Update(&key->md,out+iv,len);
460 460     SHA1_Final(mac,&key->md);
461 461
462 462
463 463     /* calculate HMAC */
464 464     key->md = key->head;
465 465     SHA1_Update(&key->md,out+iv,len);
466 466     SHA1_Final(mac,&key->md);
467 467
468 468     /* calculate HMAC */
469 469     key->md = key->head;
470 470     SHA1_Update(&key->md,out+iv,len);
471 471     SHA1_Final(mac,&key->md);
472 472
473 473     /* calculate HMAC */
474 474     key->md = key->head;
475 475     SHA1_Update(&key->md,out+iv,len);
476 476     SHA1_Final(mac,&key->md);
477 477
478 478     /* calculate HMAC */
479 479     key->md = key->head;
480 480     SHA1_Update(&key->md,out+iv,len);
481 481     SHA1_Final(mac,&key->md);
482 482
483 483     /* calculate HMAC */
484 484     key->md = key->head;
485 485     SHA1_Update(&key->md,out+iv,len);
486 486     SHA1_Final(mac,&key->md);
487 487
488 488     /* calculate HMAC */
489 489     key->md = key->head;
490 490     SHA1_Update(&key->md,out+iv,len);
491 491     SHA1_Final(mac,&key->md);
492 492
493 493     /* calculate HMAC */
494 494     key->md = key->head;
495 495     SHA1_Update(&key->md,out+iv,len);
496 496     SHA1_Final(mac,&key->md);
497 497
498 498     /* calculate HMAC */
499 499     key->md = key->head;
500 500     SHA1_Update(&key->md,out+iv,len);
501 501     SHA1_Final(mac,&key->md);
502 502
503 503     /* calculate HMAC */
504 504     key->md = key->head;
505 505     SHA1_Update(&key->md,out+iv,len);
506 506     SHA1_Final(mac,&key->md);
507 507
508 508     /* calculate HMAC */
509 509     key->md = key->head;
510 510     SHA1_Update(&key->md,out+iv,len);
511 511     SHA1_Final(mac,&key->md);
512 512
513 513     /* calculate HMAC */
514 514     key->md = key->head;
515 515     SHA1_Update(&key->md,out+iv,len);
516 516     SHA1_Final(mac,&key->md);
517 517
518 518     /* calculate HMAC */
519 519     key->md = key->head;
520 520     SHA1_Update(&key->md,out+iv,len);
521 521     SHA1_Final(mac,&key->md);
522 522
523 523     /* calculate HMAC */
524 524     key->md = key->head;
525 525     SHA1_Update(&key->md,out+iv,len);
526 526     SHA1_Final(mac,&key->md);
527 527
528 528     /* calculate HMAC */
529 529     key->md = key->head;
530 530     SHA1_Update(&key->md,out+iv,len);
531 531     SHA1_Final(mac,&key->md);
532 532
533 533     /* calculate HMAC */
534 534     key->md = key->head;
535 535     SHA1_Update(&key->md,out+iv,len);
536 536     SHA1_Final(mac,&key->md);
537 537
538 538     /* calculate HMAC */
539 539     key->md = key->head;
540 540     SHA1_Update(&key->md,out+iv,len);
541 541     SHA1_Final(mac,&key->md);
542 542
543 543     /* calculate HMAC */
544 544     key->md = key->head;
545 545     SHA1_Update(&key->md,out+iv,len);
546 546     SHA1_Final(mac,&key->md);
547 547
548 548     /* calculate HMAC */
549 549     key->md = key->head;
550 550     SHA1_Update(&key->md,out+iv,len);
551 551     SHA1_Final(mac,&key->md);
552 552
553 553     /* calculate HMAC */
554 554     key->md = key->head;
555 555     SHA1_Update(&key->md,out+iv,len);
556 556     SHA1_Final(mac,&key->md);
557 557
558 558     /* calculate HMAC */
559 559     key->md = key->head;
560 560     SHA1_Update(&key->md,out+iv,len);
561 561     SHA1_Final(mac,&key->md);
562 562
563 563     /* calculate HMAC */
564 564     key->md = key->head;
565 565     SHA1_Update(&key->md,out+iv,len);
566 566     SHA1_Final(mac,&key->md);
567 567
568 568     /* calculate HMAC */
569 569     key->md = key->head;
570 570     SHA1_Update(&key->md,out+iv,len);
571 571     SHA1_Final(mac,&key->md);
572 572
573 573     /* calculate HMAC */
574 574     key->md = key->head;
575 575     SHA1_Update(&key->md,out+iv,len);
576 576     SHA1_Final(mac,&key->md);
577 577
578 578     /* calculate HMAC */
579 579     key->md = key->head;
580 580     SHA1_Update(&key->md,out+iv,len);
581 581     SHA1_Final(mac,&key->md);
582 582
583 583     maxpad |= (255 - maxpad) >> (sizeof(maxpad) * 8 - 8);
584 584     maxpad &= 255;
585 585
586 586     ret &= constant_time_ge(maxpad, pad);
587 587
588 588     inp_len = len - (SHA_DIGEST_LENGTH + pad + 1);
589 589     mask = (0 - ((inp_len - len) >> (sizeof(inp_len) * 8 - 1)));
590 590     inp_len &= mask;
591 591
592 592     mask = (0-((inp_len-len)>>(sizeof(inp_len)*8-1)));
593 593     inp_len &= mask;
594 594     ret &= (int)mask;
595 595
596 596     /* calculate HMAC and verify it */
597 597     key->aux.tls_aad[plen-2] = inp_len>>8;
598 598     key->aux.tls_aad[plen-1] = inp_len;
599 599
600 600     /* calculate HMAC */
601 601     key->md = key->head;
602 602     SHA1_Update(&key->md,key->aux.tls_aad,plen);
603 603     SHA1_Update(&key->md,out+iv,len);
604 604     SHA1_Final(mac,&key->md);
605 605
606 606     /* amend mac */
607 607     if (len>(256*SHA_CBLOCK)) {
608 608         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
609 609         j += SHA_CBLOCK-key->md.num;
610 610         SHA1_Update(&key->md,out+j);
611 611
612 612     }
613 613
614 614     /* amend mac */
615 615     key->md = key->head;
616 616     SHA1_Update(&key->md,out+iv,len);
617 617     SHA1_Final(mac,&key->md);
618 618
619 619     /* amend mac */
620 620     key->md = key->head;
621 621     SHA1_Update(&key->md,out+iv,len);
622 622     SHA1_Final(mac,&key->md);
623 623
624 624     /* amend mac */
625 625     key->md = key->head;
626 626     SHA1_Update(&key->md,out+iv,len);
627 627     SHA1_Final(mac,&key->md);
628 628
629 629     /* amend mac */
630 630     key->md = key->head;
631 631     SHA1_Update(&key->md,out+iv,len);
632 632     SHA1_Final(mac,&key->md);
633 633
634 634     /* amend mac */
635 635     key->md = key->head;
636 636     SHA1_Update(&key->md,out+iv,len);
637 637     SHA1_Final(mac,&key->md);
638 638
639 639     /* amend mac */
640 640     key->md = key->head;
641 641     SHA1_Update(&key->md,out+iv,len);
642 642     SHA1_Final(mac,&key->md);
643 643
644 644     /* amend mac */
645 645     key->md = key->head;
646 646     SHA1_Update(&key->md,out+iv,len);
647 647     SHA1_Final(mac,&key->md);
648 648
649 649     /* amend mac */
650 650     key->md = key->head;
651 651     SHA1_Update(&key->md,out+iv,len);
652 652     SHA1_Final(mac,&key->md);
653 653
654 654     /* amend mac */
655 655     key->md = key->head;
656 656     SHA1_Update(&key->md,out+iv,len);
657 657     SHA1_Final(mac,&key->md);
658 658
659 659     /* amend mac */
660 660     key->md = key->head;
661 661     SHA1_Update(&key->md,out+iv,len);
662 662     SHA1_Final(mac,&key->md);
663 663
664 664     /* amend mac */
665 665     key->md = key->head;
666 666     SHA1_Update(&key->md,out+iv,len);
667 667     SHA1_Final(mac,&key->md);
668 668
669 669     /* amend mac */
670 670     key->md = key->head;
671 671     SHA1_Update(&key->md,out+iv,len);
672 672     SHA1_Final(mac,&key->md);
673 673
674 674     /* amend mac */
675 675     key->md = key->head;
676 676     SHA1_Update(&key->md,out+iv,len);
677 677     SHA1_Final(mac,&key->md);
678 678
679 679     /* amend mac */
680 680     key->md = key->head;
681 681     SHA1_Update(&key->md,out+iv,len);
682 682     SHA1_Final(mac,&key->md);
683 683
684 684     /* amend mac */
685 685     key->md = key->head;
686 686     SHA1_Update(&key->md,out+iv,len);
687 687     SHA1_Final(mac,&key->md);
688 688
689 689     /* amend mac */
690 690     key->md = key->head;
691 691     SHA1_Update(&key->md,out+iv,len);
692 692     SHA1_Final(mac,&key->md);
693 693
694 694     /* amend mac */
695 695     key->md = key->head;
696 696     SHA1_Update(&key->md,out+iv,len);
697 697     SHA1_Final(mac,&key->md);
698 698
699 699     /* amend mac */
700 700     key->md = key->head;
701 701     SHA1_Update(&key->md,out+iv,len);
702 702     SHA1_Final(mac,&key->md);
703 703
704 704     /* amend mac */
705 705     key->md = key->head;
706 706     SHA1_Update(&key->md,out+iv,len);
707 707     SHA1_Final(mac,&key->md);
708 708
709 709     /* amend mac */
710 710     key->md = key->head;
711 711     SHA1_Update(&key->md,out+iv,len);
712 712     SHA1_Final(mac,&key->md);
713 713
714 714     /* amend mac */
715 715     key->md = key->head;
716 716     SHA1_Update(&key->md,out+iv,len);
717 717     SHA1_Final(mac,&key->md);
718 718
719 719     /* amend mac */
720 720     key->md = key->head;
721 721     SHA1_Update(&key->md,out+iv,len);
722 722     SHA1_Final(mac,&key->md);
723 723
724 724     /* amend mac */
725 725     key->md = key->head;
726 726     SHA1_Update(&key->md,out+iv,len);
727 727     SHA1_Final(mac,&key->md);
728 728
729 729     /* amend mac */
730 730     key->md = key->head;
731 731     SHA1_Update(&key->md,out+iv,len);
732 732     SHA1_Final(mac,&key->md);
733 733
734 734     /* amend mac */
735 735     key->md = key->head;
736 736     SHA1_Update(&key->md,out+iv,len);
737 737     SHA1_Final(mac,&key->md);
738 738
739 739     /* amend mac */
740 740     key->md = key->head;
741 741     SHA1_Update(&key->md,out+iv,len);
742 742     SHA1_Final(mac,&key->md);
743 743
744 744     /* amend mac */
745 745     key->md = key->head;
746 746     SHA1_Update(&key->md,out+iv,len);
747 747     SHA1_Final(mac,&key->md);
748 748
749 749     /* amend mac */
750 750     key->md = key->head;
751 751     SHA1_Update(&key->md,out+iv,len);
752 752     SHA1_Final(mac,&key->md);
753 753
754 754     /* amend mac */
755 755     key->md = key->head;
756 756     SHA1_Update(&key->md,out+iv,len);
757 757     SHA1_Final(mac,&key->md);
758 758
759 759     /* amend mac */
760 760     key->md = key->head;
761 761     SHA1_Update(&key->md,out+iv,len);
762 762     SHA1_Final(mac,&key->md);
763 763
764 764     /* amend mac */
765 765     key->md = key->head;
766 766     SHA1_Update(&key->md,out+iv,len);
767 767     SHA1_Final(mac,&key->md);
768 768
769 769     /* amend mac */
770 770     key->md = key->head;
771 771     SHA1_Update(&key->md,out+iv,len);
772 772     SHA1_Final(mac,&key->md);
773 773
774 774     /* amend mac */
775 775     key->md = key->head;
776 776     SHA1_Update(&key->md,out+iv,len);
777 777     SHA1_Final(mac,&key->md);
778 778
779 779     /* amend mac */
780 780     key->md = key->head;
781 781     SHA1_Update(&key->md,out+iv,len);
782 782     SHA1_Final(mac,&key->md);
783 783
784 784     /* amend mac */
785 785     key->md = key->head;
786 786     SHA1_Update(&key->md,out+iv,len);
787 787     SHA1_Final(mac,&key->md);
788 788
789 789     /* amend mac */
790 790     key->md = key->head;
791 791     SHA1_Update(&key->md,out+iv,len);
792 792     SHA1_Final(mac,&key->md);
793 793
794 794     /* amend mac */
795 795     key->md = key->head;
796 796     SHA1_Update(&key->md,out+iv,len);
797 797     SHA1_Final(mac,&key->md);
798 798
799 799     /* amend mac */
800 800     key->md = key->head;
801 801     SHA1_Update(&key->md,out+iv,len);
802 802     SHA1_Final(mac,&key->md);
803 803
804 804     /* amend mac */
805 805     key->md = key->head;
806 806     SHA1_Update(&key->md,out+iv,len);
807 807     SHA1_Final(mac,&key->md);
808 808
809 809     /* amend mac */
810 810     key->md = key->head;
811 811     SHA1_Update(&key->md,out+iv,len);
812 812     SHA1_Final(mac,&key->md);
813 813
814 814     /* amend mac */
815 815     key->md = key->head;
816 816     SHA1_Update(&key->md,out+iv,len);
817 817     SHA1_Final(mac,&key->md);
818 818
819 819     /* amend mac */
820 820     key->md = key->head;
821 821     SHA1_Update(&key->md,out+iv,len);
822 822     SHA1_Final(mac,&key->md);
823 823
824 824     /* amend mac */
825 825     key->md = key->head;
826 826     SHA1_Update(&key->md,out+iv,len);
827 827     SHA1_Final(mac,&key->md);
828 828
829 829     /* amend mac */
830 830     key->md = key->head;
831 831     SHA1_Update(&key->md,out+iv,len);
832 832     SHA1_Final(mac,&key->md);
833 833
834 834     /* amend mac */
835 835     key->md = key->head;
836 836     SHA1_Update(&key->md,out+iv,len);
837 837     SHA1_Final(mac,&key->md);
838 838
839 839     /* amend mac */
840 840     key->md = key->head;
841 841     SHA1_Update(&key->md,out+iv,len);
842 842     SHA1_Final(mac,&key->md);
843 843
844 844     /* amend mac */
845 845     key->md = key->head;
846 846     SHA1_Update(&key->md,out+iv,len);
847 847     SHA1_Final(mac,&key->md);
848 848
849 849     /* amend mac */
850 850     key->md = key->head;
851 851     SHA1_Update(&key->md,out+iv,len);
852 852     SHA1_Final(mac,&key->md);
853 853
854 854     /* amend mac */
855 855     key->md = key->head;
856 856     SHA1_Update(&key->md,out+iv,len);
857 857     SHA1_Final(mac,&key->md);
858 858
859 859     /* amend mac */
860 860     key->md = key->head;
861 861     SHA1_Update(&key->md,out+iv,len);
862 862     SHA1_Final(mac,&key->md);
863 863
864 864     /* amend mac */
865 865     key->md = key->head;
866 866     SHA1_Update(&key->md,out+iv,len);
867 867     SHA1_Final(mac,&key->md);
868 868
869 869     /* amend mac */
870 870     key->md = key->head;
871 871     SHA1_Update(&key->md,out+iv,len);
872 872     SHA1_Final(mac,&key->md);
873 873
874 87
```

Writing CT code is unholy

The screenshot shows a code editor displaying a portion of the OpenSSL source code. The code is color-coded by file: green for C code, red for comments, and blue for preprocessor directives. Several sections of the code are highlighted with different colors (pink, light green, light red) and framed by black or red lines, likely indicating specific parts of interest for analysis. The code itself is a complex cryptographic routine involving EVP_DigestUpdate, EVP_DigestFinal_ex, and SHA functions.

```
380 383     SSL3_RECORD *rr;
384 384     unsigned int mac_size;
385 385     unsigned char md[EVP_MAX_MD_SIZE];
386 386 +     int decryption_failed_or_bad_record_mac = 0;
387 387
388 388     EVP_DigestUpdate(&md_ctx, md, 2);
389 389     EVP_DigestUpdate(&md_ctx, rec->input, rec->length);
390 390     EVP_DigestFinal_ex( &md_ctx, md, NULL);
391 391
392 392     unsigned char mac[SHA_DIGEST_LENGTH];
393 393     union { unsigned int u[SHA_DIGEST_LENGTH/sizeof(unsigned int)];
394 394         unsigned char c[SHA_DIGEST_LENGTH]; } mac;
395 395
396 396     /* decrypt HMAC(padding at once */
397 397     aesni_cbc_encrypt(in,out,len,
398 398         &key->ks,ctx->iv,0);
399 399
400 400     if (plen) { /* "TLS" mode of operation */
401 401         /* figure out payload length */
402 402         if (len<(size_t)(out[len-1]+SHA_DIGEST_LENGTH))
403 403             return 0;
404 404
405 405         len -= (out[len-1]+SHA_DIGEST_LENGTH);
406 406         size_t inp_len, mask, j, 1;
407 407         unsigned int res, maxpad, pad, bitlen;
408 408         int ret = 1;
409 409         union { unsigned int u[SHA_CBLOCK];
410 410             unsigned char c[SHA_CBLOCK]; }
411 411         *data = (void *)key->md.data;
412 412
413 413         if ((key->aux.tls_aad[plen-4]<=8||key->aux.tls_aad[plen-3])
414 414             >= TLS1_1_VERSION) {
415 415             len -= AES_BLOCK_SIZE;
416 416             >= TLS1_1_VERSION
417 417             iv = AES_BLOCK_SIZE;
418 418         }
419 419
420 420         key->aux.tls_aad[plen-2] = len>>8;
421 421         key->aux.tls_aad[plen-1] = len;
422 422         /* calculate HMAC and verify it */
423 423
424 424     maxpad |= (255 - maxpad) >> (sizeof(maxpad) * 8 - 8);
425 425     maxpad &= 255;
426 426
427 427     ret &= constant_time_ge(maxpad, pad);
428 428
429 429     inp_len = len - (SHA_DIGEST_LENGTH + pad + 1);
430 430     mask = (0 - ((inp_len - len) >> (sizeof(inp_len) * 8 - 1)));
431 431     inp_len &= mask;
432 432
433 433     mask = (0-((inp_len-len)>>(sizeof(inp_len)*8-1)));
434 434     inp_len &= mask;
435 435     ret &= (int)mask;
436 436
437 437     /* calculate HMAC and verify it */
438 438     key->md = key->head;
439 439     SHA1_Update(&key->md, key->aux.tls_aad, plen);
440 440     SHA1_Update(&key->md, out+iv, len);
441 441     SHA1_Final(mac,&key->md);
442 442
443 443     /* amend mac */
444 444     if (len>=(256*SHA_CBLOCK)) {
445 445         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
446 446         j += SHA_CBLOCK-key->md.num;
447 447         SHA1_Update(&key->md, out+j);
448 448
449 449     len -= SHA_DIGEST_LENGTH;
450 450     if (len>=(256*SHA_CBLOCK)) {
451 451         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
452 452         j += SHA_CBLOCK-key->md.num;
453 453         SHA1_Update(&key->md, out+j);
454 454
455 455     len -= SHA_DIGEST_LENGTH;
456 456     if (len>=(256*SHA_CBLOCK)) {
457 457         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
458 458         j += SHA_CBLOCK-key->md.num;
459 459         SHA1_Update(&key->md, out+j);
460 460
461 461     len -= SHA_DIGEST_LENGTH;
462 462     if (len>=(256*SHA_CBLOCK)) {
463 463         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
464 464         j += SHA_CBLOCK-key->md.num;
465 465         SHA1_Update(&key->md, out+j);
466 466
467 467     len -= SHA_DIGEST_LENGTH;
468 468     if (len>=(256*SHA_CBLOCK)) {
469 469         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
470 470         j += SHA_CBLOCK-key->md.num;
471 471         SHA1_Update(&key->md, out+j);
472 472
473 473     len -= SHA_DIGEST_LENGTH;
474 474     if (len>=(256*SHA_CBLOCK)) {
475 475         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
476 476         j += SHA_CBLOCK-key->md.num;
477 477         SHA1_Update(&key->md, out+j);
478 478
479 479     len -= SHA_DIGEST_LENGTH;
480 480     if (len>=(256*SHA_CBLOCK)) {
481 481         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
482 482         j += SHA_CBLOCK-key->md.num;
483 483         SHA1_Update(&key->md, out+j);
484 484
485 485     len -= SHA_DIGEST_LENGTH;
486 486     if (len>=(256*SHA_CBLOCK)) {
487 487         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
488 488         j += SHA_CBLOCK-key->md.num;
489 489         SHA1_Update(&key->md, out+j);
490 490
491 491     len -= SHA_DIGEST_LENGTH;
492 492     if (len>=(256*SHA_CBLOCK)) {
493 493         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
494 494         j += SHA_CBLOCK-key->md.num;
495 495         SHA1_Update(&key->md, out+j);
496 496
497 497     len -= SHA_DIGEST_LENGTH;
498 498     if (len>=(256*SHA_CBLOCK)) {
499 499         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
500 500         j += SHA_CBLOCK-key->md.num;
501 501         SHA1_Update(&key->md, out+j);
502 502
503 503     len -= SHA_DIGEST_LENGTH;
504 504     if (len>=(256*SHA_CBLOCK)) {
505 505         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
506 506         j += SHA_CBLOCK-key->md.num;
507 507         SHA1_Update(&key->md, out+j);
508 508
509 509     len -= SHA_DIGEST_LENGTH;
510 510     if (len>=(256*SHA_CBLOCK)) {
511 511         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
512 512         j += SHA_CBLOCK-key->md.num;
513 513         SHA1_Update(&key->md, out+j);
514 514
515 515     len -= SHA_DIGEST_LENGTH;
516 516     if (len>=(256*SHA_CBLOCK)) {
517 517         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
518 518         j += SHA_CBLOCK-key->md.num;
519 519         SHA1_Update(&key->md, out+j);
520 520
521 521     len -= SHA_DIGEST_LENGTH;
522 522     if (len>=(256*SHA_CBLOCK)) {
523 523         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
524 524         j += SHA_CBLOCK-key->md.num;
525 525         SHA1_Update(&key->md, out+j);
526 526
527 527     len -= SHA_DIGEST_LENGTH;
528 528     if (len>=(256*SHA_CBLOCK)) {
529 529         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
530 530         j += SHA_CBLOCK-key->md.num;
531 531         SHA1_Update(&key->md, out+j);
532 532
533 533     len -= SHA_DIGEST_LENGTH;
534 534     if (len>=(256*SHA_CBLOCK)) {
535 535         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
536 536         j += SHA_CBLOCK-key->md.num;
537 537         SHA1_Update(&key->md, out+j);
538 538
539 539     len -= SHA_DIGEST_LENGTH;
540 540     if (len>=(256*SHA_CBLOCK)) {
541 541         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
542 542         j += SHA_CBLOCK-key->md.num;
543 543         SHA1_Update(&key->md, out+j);
544 544
545 545     len -= SHA_DIGEST_LENGTH;
546 546     if (len>=(256*SHA_CBLOCK)) {
547 547         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
548 548         j += SHA_CBLOCK-key->md.num;
549 549         SHA1_Update(&key->md, out+j);
550 550
551 551     len -= SHA_DIGEST_LENGTH;
552 552     if (len>=(256*SHA_CBLOCK)) {
553 553         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
554 554         j += SHA_CBLOCK-key->md.num;
555 555         SHA1_Update(&key->md, out+j);
556 556
557 557     len -= SHA_DIGEST_LENGTH;
558 558     if (len>=(256*SHA_CBLOCK)) {
559 559         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
560 560         j += SHA_CBLOCK-key->md.num;
561 561         SHA1_Update(&key->md, out+j);
562 562
563 563     len -= SHA_DIGEST_LENGTH;
564 564     if (len>=(256*SHA_CBLOCK)) {
565 565         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
566 566         j += SHA_CBLOCK-key->md.num;
567 567         SHA1_Update(&key->md, out+j);
568 568
569 569     len -= SHA_DIGEST_LENGTH;
570 570     if (len>=(256*SHA_CBLOCK)) {
571 571         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
572 572         j += SHA_CBLOCK-key->md.num;
573 573         SHA1_Update(&key->md, out+j);
574 574
575 575     len -= SHA_DIGEST_LENGTH;
576 576     if (len>=(256*SHA_CBLOCK)) {
577 577         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
578 578         j += SHA_CBLOCK-key->md.num;
579 579         SHA1_Update(&key->md, out+j);
580 580
581 581     len -= SHA_DIGEST_LENGTH;
582 582     if (len>=(256*SHA_CBLOCK)) {
583 583         j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
584 584         j += SHA_CBLOCK-key->md.num;
585 585         SHA1_Update(&key->md, out+j);
586 586
587 587         ret &= constant_time_ge(maxpad, pad);
588 588
589 589         inp_len = len - (SHA_DIGEST_LENGTH + pad + 1);
590 590         mask = (0 - ((inp_len - len) >> (sizeof(inp_len) * 8 - 1)));
591 591         inp_len &= mask;
592 592
593 593         mask = (0-((inp_len-len)>>(sizeof(inp_len)*8-1)));
594 594         inp_len &= mask;
595 595         ret &= (int)mask;
596 596
597 597         /* calculate HMAC */
598 598         key->md = key->head;
599 599         SHA1_Update(&key->md, key->aux.tls_aad, plen);
600 600         SHA1_Update(&key->md, out+iv, len);
601 601         SHA1_Final(mac,&key->md);
602 602
603 603         /* amend mac */
604 604         if (len>=(256*SHA_CBLOCK)) {
605 605             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
606 606             j += SHA_CBLOCK-key->md.num;
607 607             SHA1_Update(&key->md, out+j);
608 608
609 609         len -= SHA_DIGEST_LENGTH;
610 610         if (len>=(256*SHA_CBLOCK)) {
611 611             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
612 612             j += SHA_CBLOCK-key->md.num;
613 613             SHA1_Update(&key->md, out+j);
614 614
615 615         len -= SHA_DIGEST_LENGTH;
616 616         if (len>=(256*SHA_CBLOCK)) {
617 617             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
618 618             j += SHA_CBLOCK-key->md.num;
619 619             SHA1_Update(&key->md, out+j);
620 620
621 621         len -= SHA_DIGEST_LENGTH;
622 622         if (len>=(256*SHA_CBLOCK)) {
623 623             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
624 624             j += SHA_CBLOCK-key->md.num;
625 625             SHA1_Update(&key->md, out+j);
626 626
627 627         len -= SHA_DIGEST_LENGTH;
628 628         if (len>=(256*SHA_CBLOCK)) {
629 629             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
630 630             j += SHA_CBLOCK-key->md.num;
631 631             SHA1_Update(&key->md, out+j);
632 632
633 633         len -= SHA_DIGEST_LENGTH;
634 634         if (len>=(256*SHA_CBLOCK)) {
635 635             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
636 636             j += SHA_CBLOCK-key->md.num;
637 637             SHA1_Update(&key->md, out+j);
638 638
639 639         len -= SHA_DIGEST_LENGTH;
640 640         if (len>=(256*SHA_CBLOCK)) {
641 641             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
642 642             j += SHA_CBLOCK-key->md.num;
643 643             SHA1_Update(&key->md, out+j);
644 644
645 645         len -= SHA_DIGEST_LENGTH;
646 646         if (len>=(256*SHA_CBLOCK)) {
647 647             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
648 648             j += SHA_CBLOCK-key->md.num;
649 649             SHA1_Update(&key->md, out+j);
650 650
651 651         len -= SHA_DIGEST_LENGTH;
652 652         if (len>=(256*SHA_CBLOCK)) {
653 653             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
654 654             j += SHA_CBLOCK-key->md.num;
655 655             SHA1_Update(&key->md, out+j);
656 656
657 657         len -= SHA_DIGEST_LENGTH;
658 658         if (len>=(256*SHA_CBLOCK)) {
659 659             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
660 660             j += SHA_CBLOCK-key->md.num;
661 661             SHA1_Update(&key->md, out+j);
662 662
663 663         len -= SHA_DIGEST_LENGTH;
664 664         if (len>=(256*SHA_CBLOCK)) {
665 665             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
666 666             j += SHA_CBLOCK-key->md.num;
667 667             SHA1_Update(&key->md, out+j);
668 668
669 669         len -= SHA_DIGEST_LENGTH;
670 670         if (len>=(256*SHA_CBLOCK)) {
671 671             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
672 672             j += SHA_CBLOCK-key->md.num;
673 673             SHA1_Update(&key->md, out+j);
674 674
675 675         len -= SHA_DIGEST_LENGTH;
676 676         if (len>=(256*SHA_CBLOCK)) {
677 677             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
678 678             j += SHA_CBLOCK-key->md.num;
679 679             SHA1_Update(&key->md, out+j);
680 680
681 681         len -= SHA_DIGEST_LENGTH;
682 682         if (len>=(256*SHA_CBLOCK)) {
683 683             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
684 684             j += SHA_CBLOCK-key->md.num;
685 685             SHA1_Update(&key->md, out+j);
686 686
687 687         len -= SHA_DIGEST_LENGTH;
688 688         if (len>=(256*SHA_CBLOCK)) {
689 689             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
690 690             j += SHA_CBLOCK-key->md.num;
691 691             SHA1_Update(&key->md, out+j);
692 692
693 693         len -= SHA_DIGEST_LENGTH;
694 694         if (len>=(256*SHA_CBLOCK)) {
695 695             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
696 696             j += SHA_CBLOCK-key->md.num;
697 697             SHA1_Update(&key->md, out+j);
698 698
699 699         len -= SHA_DIGEST_LENGTH;
700 700         if (len>=(256*SHA_CBLOCK)) {
701 701             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
702 702             j += SHA_CBLOCK-key->md.num;
703 703             SHA1_Update(&key->md, out+j);
704 704
705 705         len -= SHA_DIGEST_LENGTH;
706 706         if (len>=(256*SHA_CBLOCK)) {
707 707             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
708 708             j += SHA_CBLOCK-key->md.num;
709 709             SHA1_Update(&key->md, out+j);
710 710
711 711         len -= SHA_DIGEST_LENGTH;
712 712         if (len>=(256*SHA_CBLOCK)) {
713 713             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
714 714             j += SHA_CBLOCK-key->md.num;
715 715             SHA1_Update(&key->md, out+j);
716 716
717 717         len -= SHA_DIGEST_LENGTH;
718 718         if (len>=(256*SHA_CBLOCK)) {
719 719             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
720 720             j += SHA_CBLOCK-key->md.num;
721 721             SHA1_Update(&key->md, out+j);
722 722
723 723         len -= SHA_DIGEST_LENGTH;
724 724         if (len>=(256*SHA_CBLOCK)) {
725 725             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
726 726             j += SHA_CBLOCK-key->md.num;
727 727             SHA1_Update(&key->md, out+j);
728 728
729 729         len -= SHA_DIGEST_LENGTH;
730 730         if (len>=(256*SHA_CBLOCK)) {
731 731             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
732 732             j += SHA_CBLOCK-key->md.num;
733 733             SHA1_Update(&key->md, out+j);
734 734
735 735         len -= SHA_DIGEST_LENGTH;
736 736         if (len>=(256*SHA_CBLOCK)) {
737 737             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
738 738             j += SHA_CBLOCK-key->md.num;
739 739             SHA1_Update(&key->md, out+j);
740 740
741 741         len -= SHA_DIGEST_LENGTH;
742 742         if (len>=(256*SHA_CBLOCK)) {
743 743             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
744 744             j += SHA_CBLOCK-key->md.num;
745 745             SHA1_Update(&key->md, out+j);
746 746
747 747         len -= SHA_DIGEST_LENGTH;
748 748         if (len>=(256*SHA_CBLOCK)) {
749 749             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
750 750             j += SHA_CBLOCK-key->md.num;
751 751             SHA1_Update(&key->md, out+j);
752 752
753 753         len -= SHA_DIGEST_LENGTH;
754 754         if (len>=(256*SHA_CBLOCK)) {
755 755             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
756 756             j += SHA_CBLOCK-key->md.num;
757 757             SHA1_Update(&key->md, out+j);
758 758
759 759         len -= SHA_DIGEST_LENGTH;
760 760         if (len>=(256*SHA_CBLOCK)) {
761 761             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
762 762             j += SHA_CBLOCK-key->md.num;
763 763             SHA1_Update(&key->md, out+j);
764 764
765 765         len -= SHA_DIGEST_LENGTH;
766 766         if (len>=(256*SHA_CBLOCK)) {
767 767             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
768 768             j += SHA_CBLOCK-key->md.num;
769 769             SHA1_Update(&key->md, out+j);
770 770
771 771         len -= SHA_DIGEST_LENGTH;
772 772         if (len>=(256*SHA_CBLOCK)) {
773 773             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
774 774             j += SHA_CBLOCK-key->md.num;
775 775             SHA1_Update(&key->md, out+j);
776 776
777 777         len -= SHA_DIGEST_LENGTH;
778 778         if (len>=(256*SHA_CBLOCK)) {
779 779             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
780 780             j += SHA_CBLOCK-key->md.num;
781 781             SHA1_Update(&key->md, out+j);
782 782
783 783         len -= SHA_DIGEST_LENGTH;
784 784         if (len>=(256*SHA_CBLOCK)) {
785 785             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
786 786             j += SHA_CBLOCK-key->md.num;
787 787             SHA1_Update(&key->md, out+j);
788 788
789 789         len -= SHA_DIGEST_LENGTH;
790 790         if (len>=(256*SHA_CBLOCK)) {
791 791             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
792 792             j += SHA_CBLOCK-key->md.num;
793 793             SHA1_Update(&key->md, out+j);
794 794
795 795         len -= SHA_DIGEST_LENGTH;
796 796         if (len>=(256*SHA_CBLOCK)) {
797 797             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
798 798             j += SHA_CBLOCK-key->md.num;
799 799             SHA1_Update(&key->md, out+j);
800 800
801 801         len -= SHA_DIGEST_LENGTH;
802 802         if (len>=(256*SHA_CBLOCK)) {
803 803             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
804 804             j += SHA_CBLOCK-key->md.num;
805 805             SHA1_Update(&key->md, out+j);
806 806
807 807         len -= SHA_DIGEST_LENGTH;
808 808         if (len>=(256*SHA_CBLOCK)) {
809 809             j = (len-(256*SHA_CBLOCK))&(0-SHA_CBLOCK);
810 810             j += SHA_CBLOCK-key->md.num;
811 811             SHA1_Update(&key->md, out+j);
812 812
813 813         len -= SHA_DIGEST_LENGTH;
814 814         if (len>=(256*SHA_CBLOCK)) {
815 815             j = (len-(256*SHA_CBLOCK
```

What can we do about this?

- Design new programming languages!
 - E.g., FaCT language lets you write code that is guaranteed to be constant time

```
export
void get_zeros_padding( secret uint8 input[], secret mut uint32 data_len)
{
    data_len = 0;
    for( uint32 i = len input; i > 0; i-=1 ) {
        if (input[i-1] != 0) {
            data_len = i;
            return;
        }
    }
}
```

Automatically transform code when possible!

```
export
void conditional_swap(secret mut uint32 x,
                      secret mut uint32 y,
                      secret bool cond) {
    if (cond) {
        secret uint32 tmp = x;
        x = y;
        y = tmp;
    }
}
```

↓

```
export
void conditional_swap(secret mut uint32 x,
                      secret mut uint32 y,
                      secret bool cond) {
    secret mut bool __branch1 = cond;
    { // then part
        secret uint32 tmp = x;
        x = CT_SEL(__branch1, y, x);
        y = CT_SEL(__branch1, tmp, y);
    }
    __branch1 = !__branch1;
    {... else part ...}
}
```

Raise type error otherwise!

- Some transformations not possible
 - E.g., loops bounded by secret data
- Some transformations would produce slow code
 - E.g., accessing array at secret index

What about existing code?

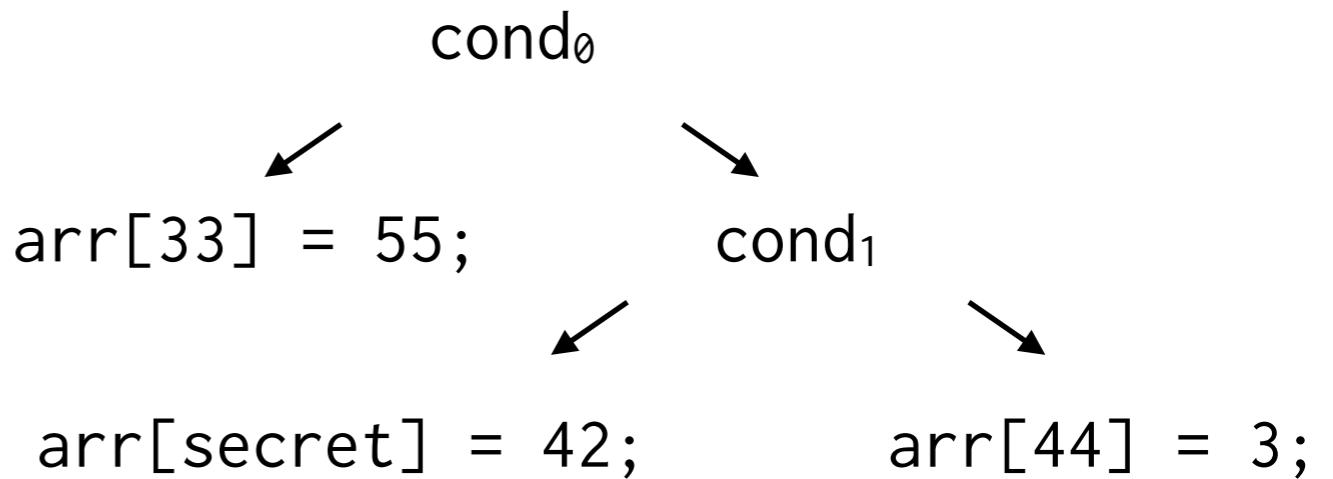
- Program analysis to the rescue!
 - Symbolically execute a function and alert if you find an input that causes it to (1) branch on a secret (2) use secret as memory index or (3) use variable-time instruction

```
if (cond0) {  
    arr[33] = 5;  
} else {  
    if (cond1) {  
        arr[secret] = 42;  
    } else {  
        arr[44] = 3;  
    }  
}
```

What about existing code?

- Program analysis to the rescue!
 - Symbolically execute a function and alert if you find an input that causes it to (1) branch on a secret (2) use secret as memory index or (3) use variable-time instruction

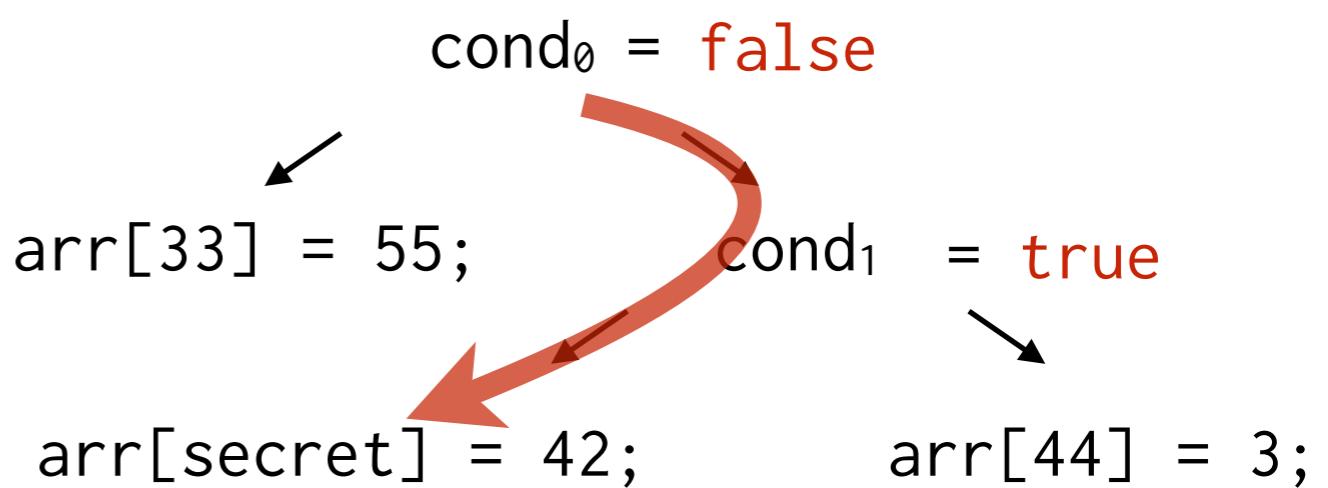
```
if (cond0) {  
    arr[33] = 5;  
} else {  
    if (cond1) {  
        arr[secret] = 42;  
    } else {  
        arr[44] = 3;  
    }  
}
```



What about existing code?

- Program analysis to the rescue!
 - Symbolically execute a function and alert if you find an input that causes it to (1) branch on a secret (2) use secret as memory index or (3) use variable-time instruction

```
if (cond0) {  
    arr[33] = 5;  
} else {  
    if (cond1) {  
        arr[secret] = 42;  
    } else {  
        arr[44] = 3;  
    }  
}
```



What about existing code?

- Program analysis to the rescue!
 - Symbolically execute a function and alert if you find an input that causes it to (1) branch on a secret (2) use secret as memory index or (3) use variable-time instruction
- Can even do this for branches not taken and find leaks via speculative execution!

