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 secretinformation 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

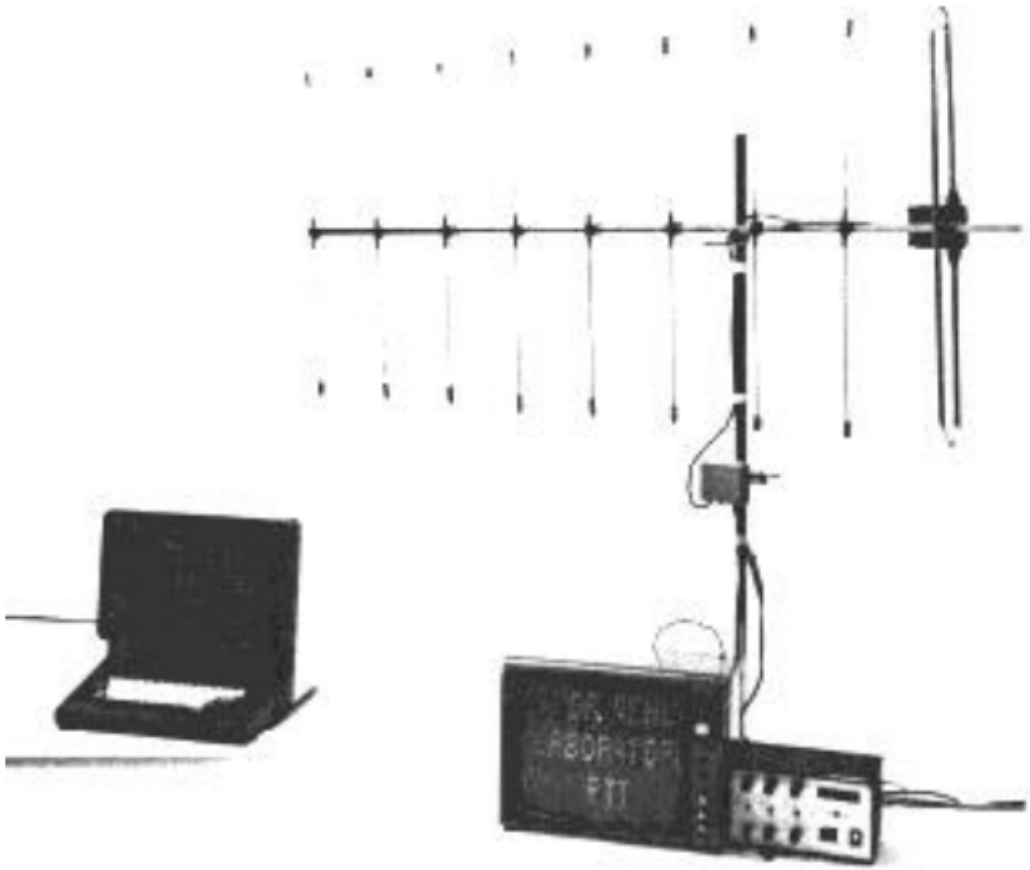
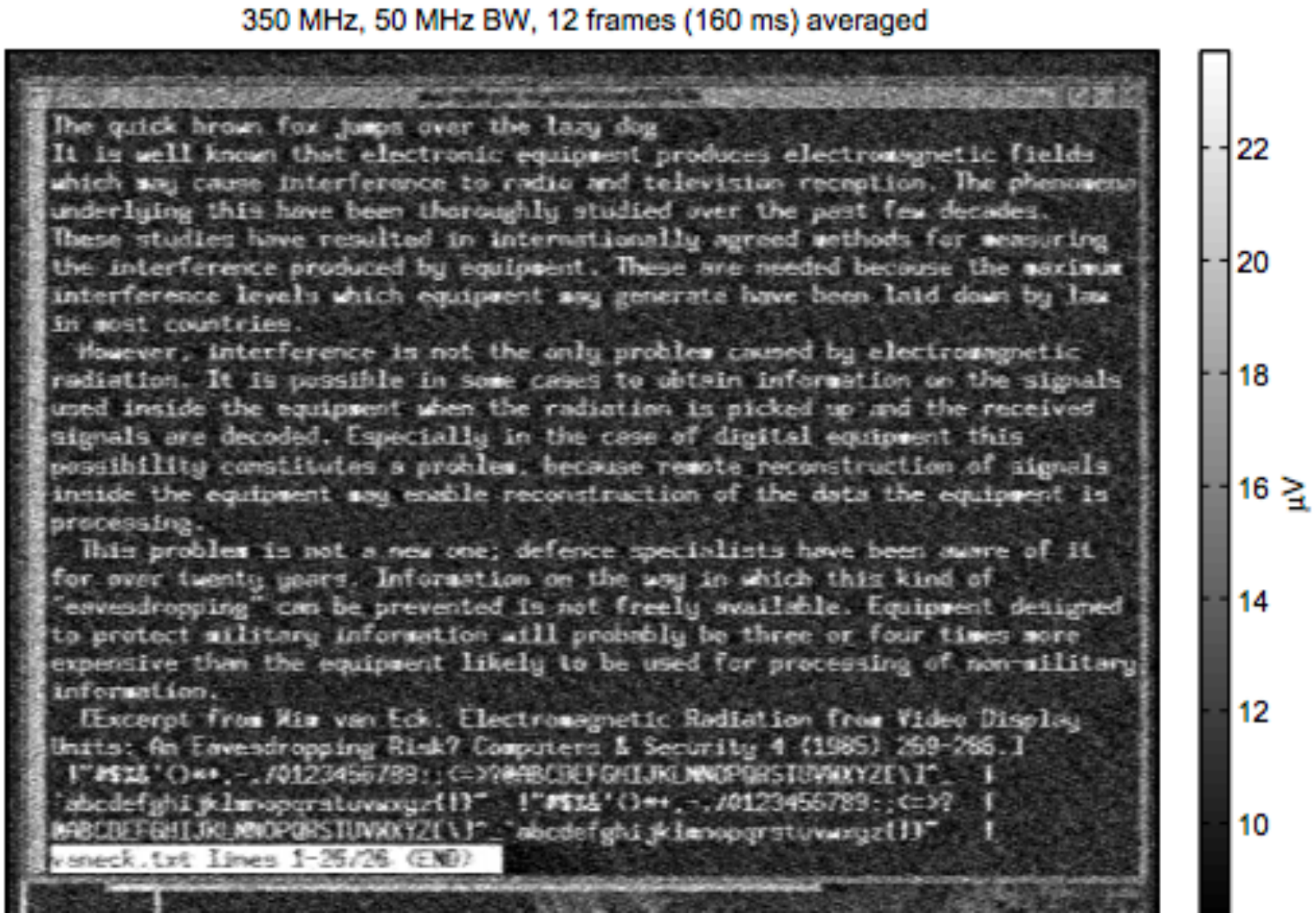


Fig. 1. Eavesdropping eel-up using a variable oscillator and a frequency divider to restore synchronization. The picture on the TV is picked up from the radiation of the VDU in the background.

“Electromagnetic Eavesdropping Risks of Flat-Panel Displays” Kuhn 2004

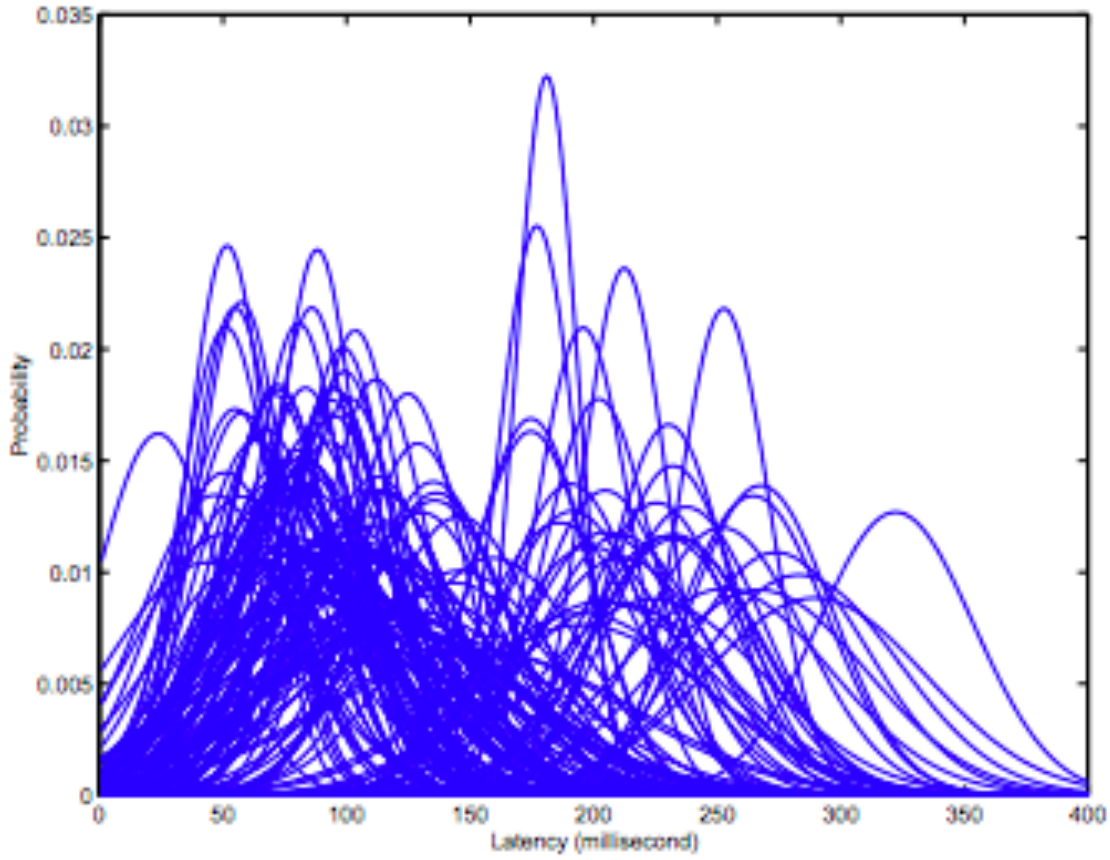
- Image displays simultaneously along line
- Pick up radiation from screen connection cable



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 multply implementation clearly leaks secret key bits in a power trace.

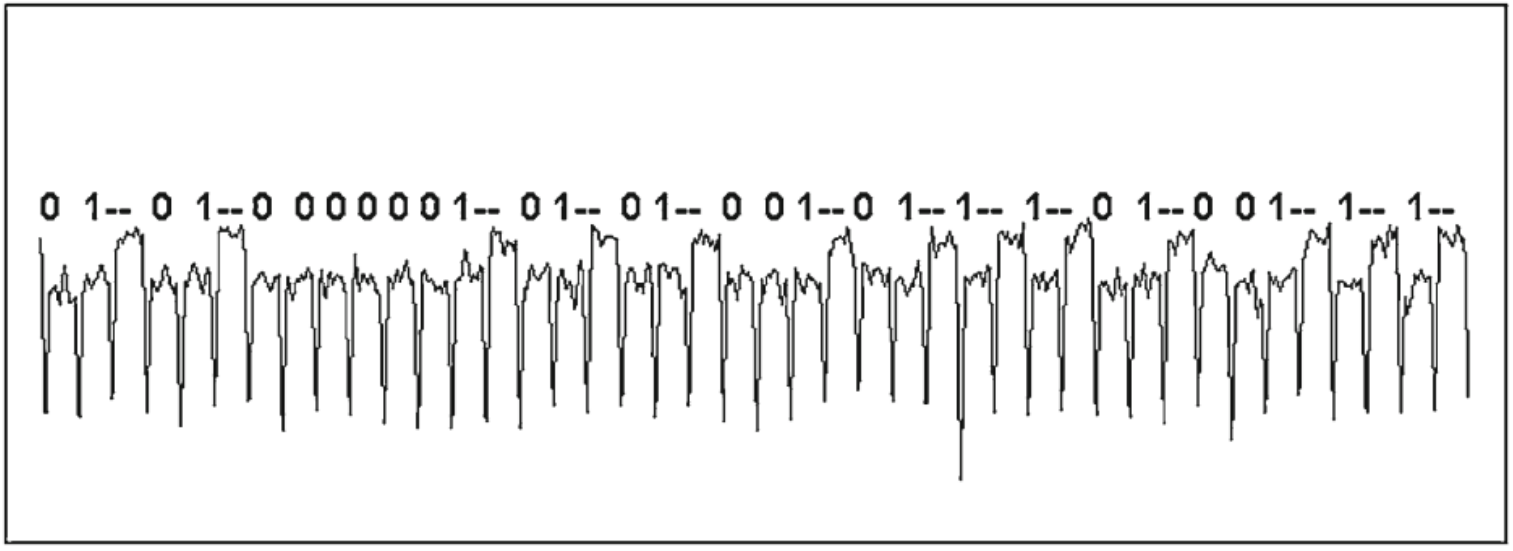
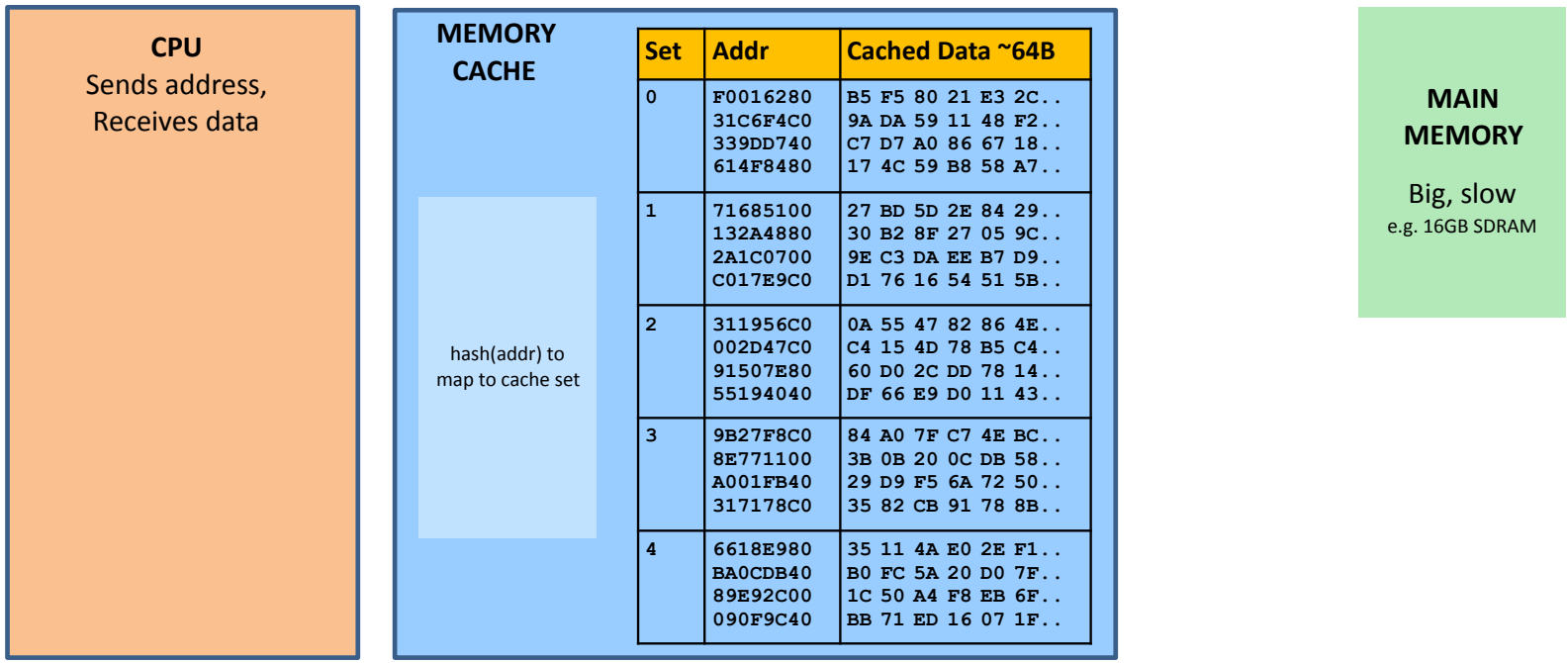


Fig. 11 SPA leaks from an RSA implementation

Reminder: Memory caches and cache attacks

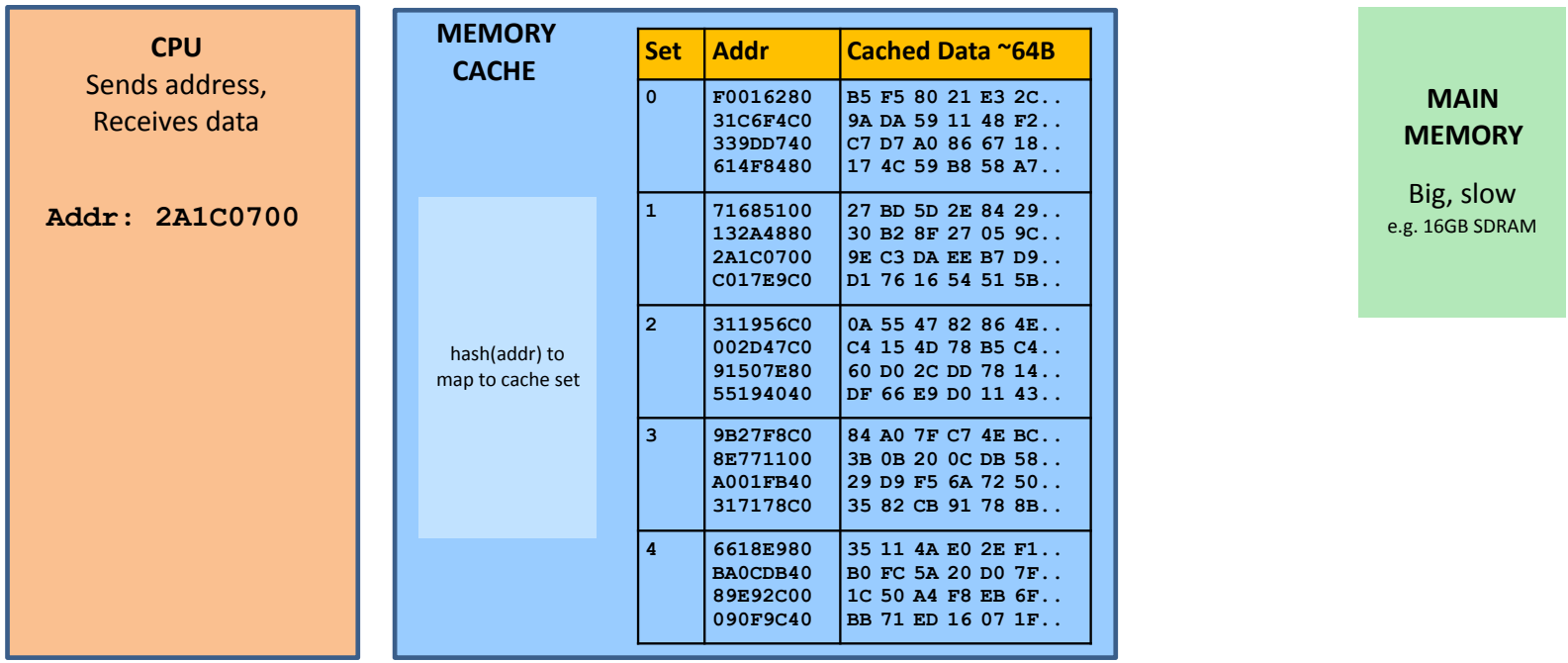
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

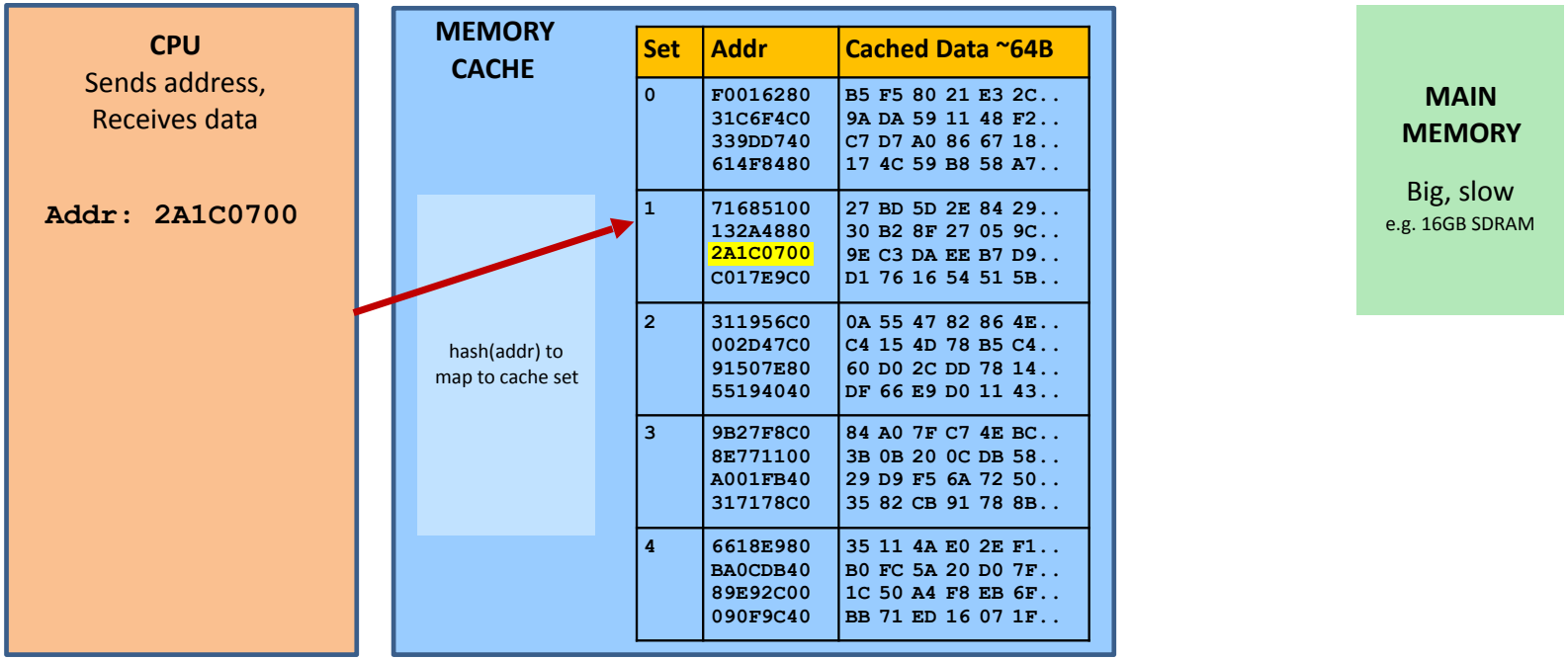
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

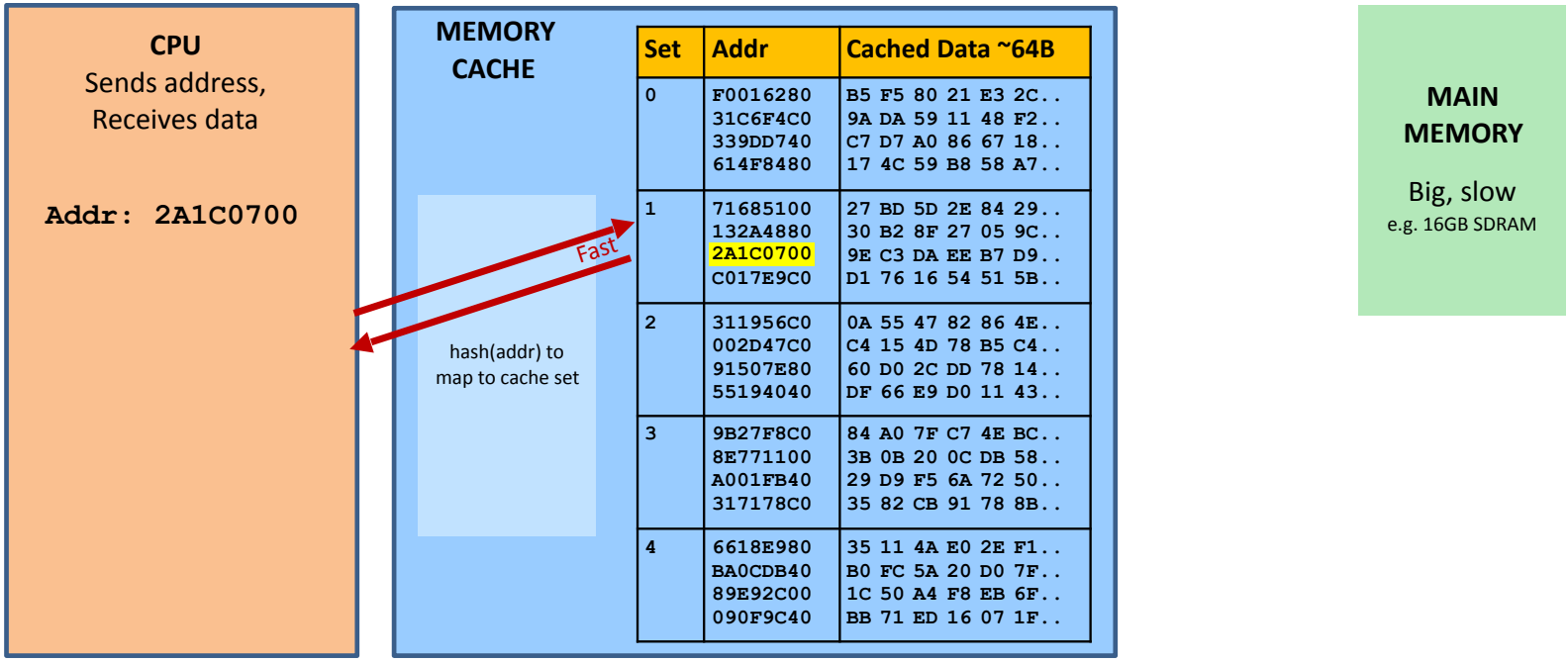
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

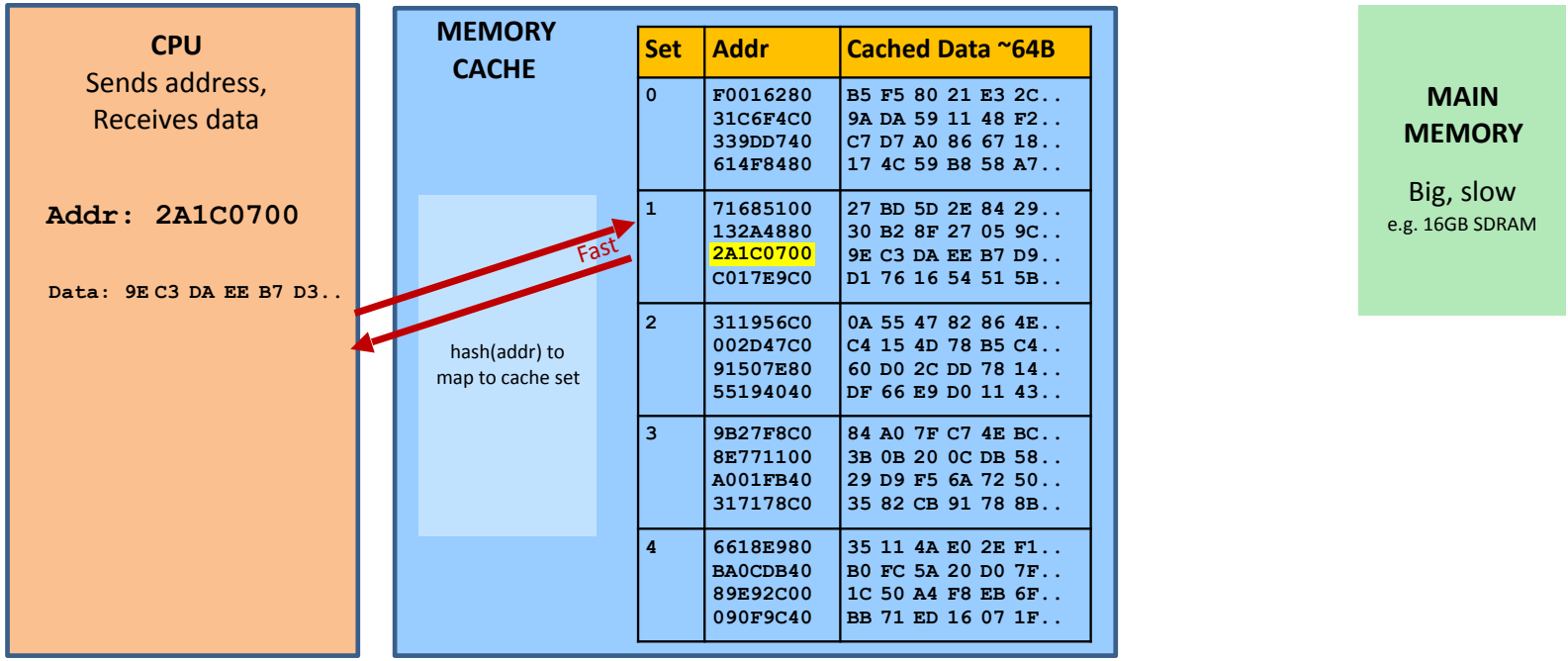


Set	Addr	Cached Data ~64B
0	F0016280 31C6F4C0 339DD740 614F8480	B5 F5 80 21 E3 2C.. 9A DA 59 11 48 F2.. C7 D7 A0 86 67 18.. 17 4C 59 B8 58 A7..
1	71685100 132A4880 2A1C0700 C017E9C0	27 BD 5D 2E 84 29.. 30 B2 8F 27 05 9C.. 9E C3 DA EE B7 D9.. D1 76 16 54 51 5B..
2	311956C0 002D47C0 91507E80 55194040	0A 55 47 82 86 4E.. C4 15 4D 78 B5 C4.. 60 D0 2C DD 78 14.. DF 66 E9 D0 11 43..
3	9B27F8C0 8E771100 A001FB40 317178C0	84 A0 7F C7 4E BC.. 3B 0B 20 0C DB 58.. 29 D9 F5 6A 72 50.. 35 82 CB 91 78 8B..
4	6618E980 BA0CDB40 89E92C00 090F9C40	35 11 4A E0 2E F1.. B0 FC 5A 20 D0 7F.. 1C 50 A4 F8 EB 6F.. BB 71 ED 16 07 1F..

- 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

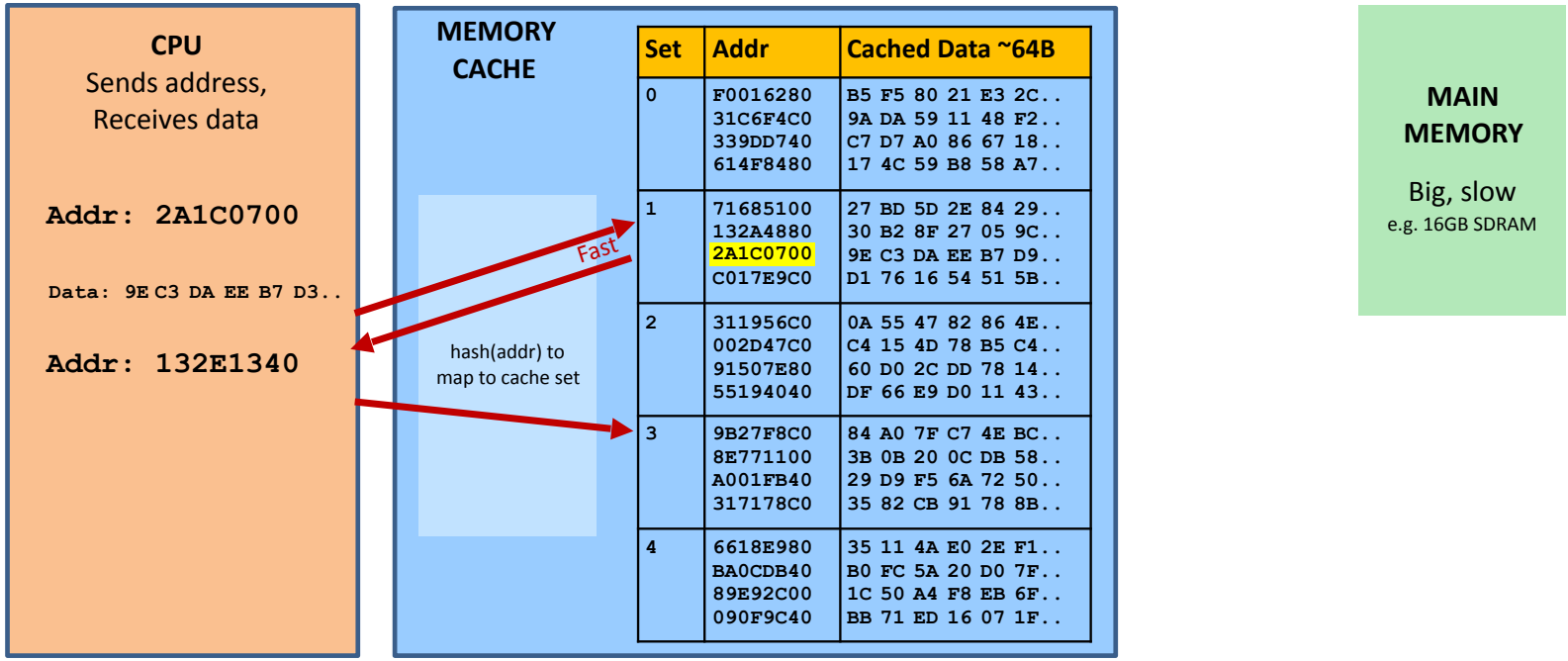
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

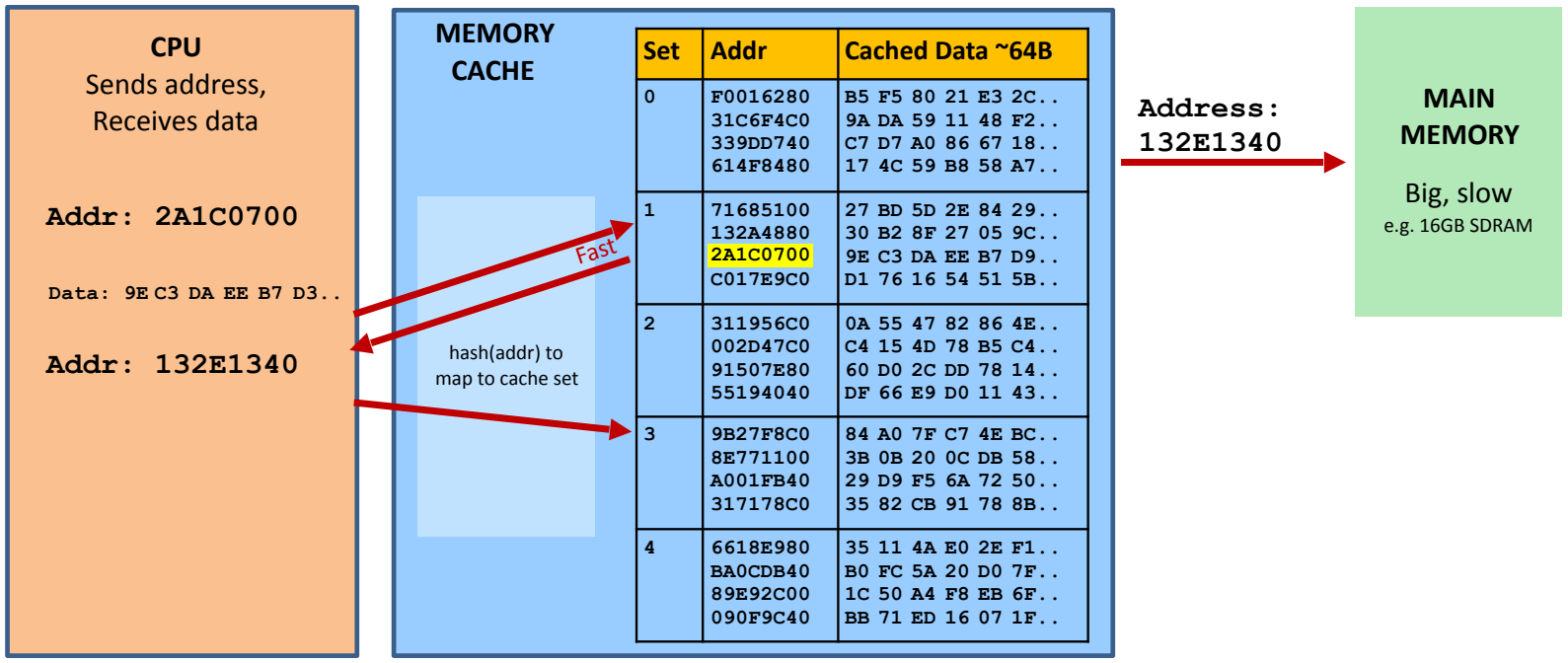
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

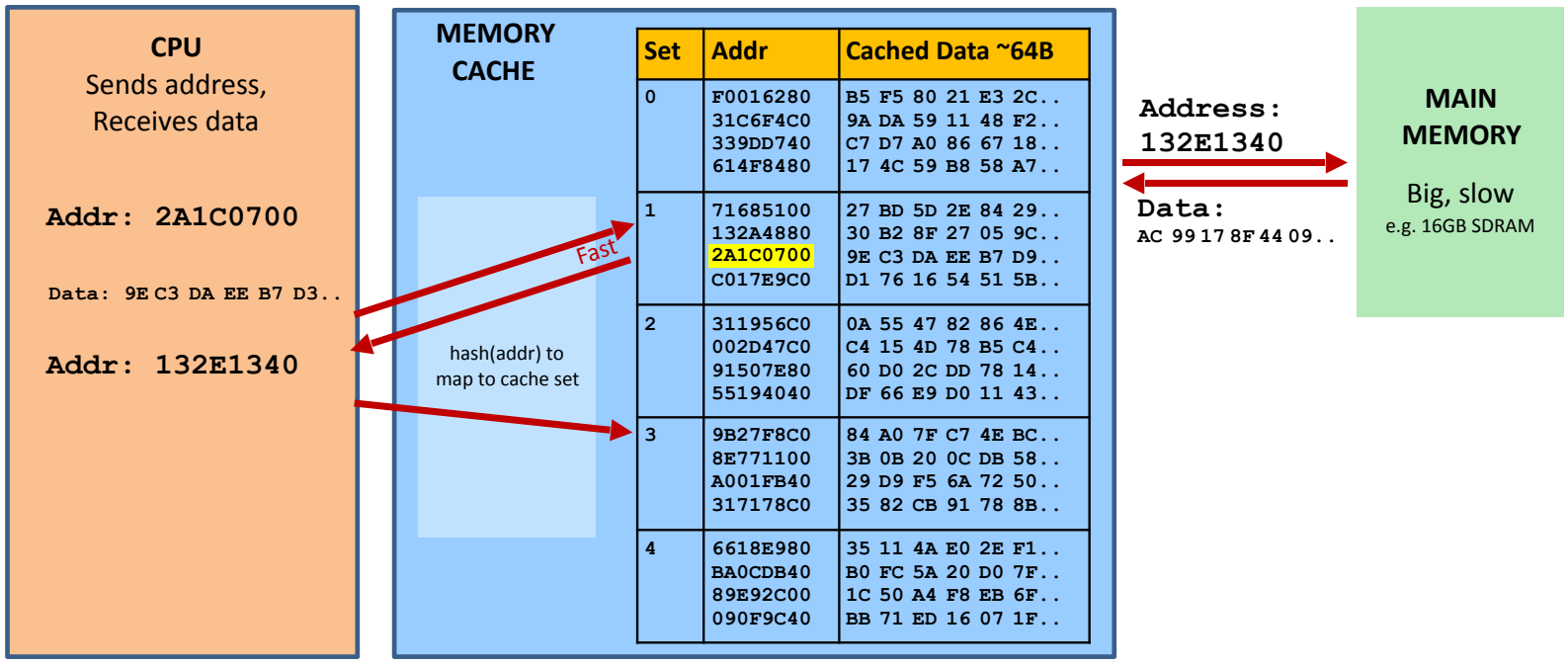
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

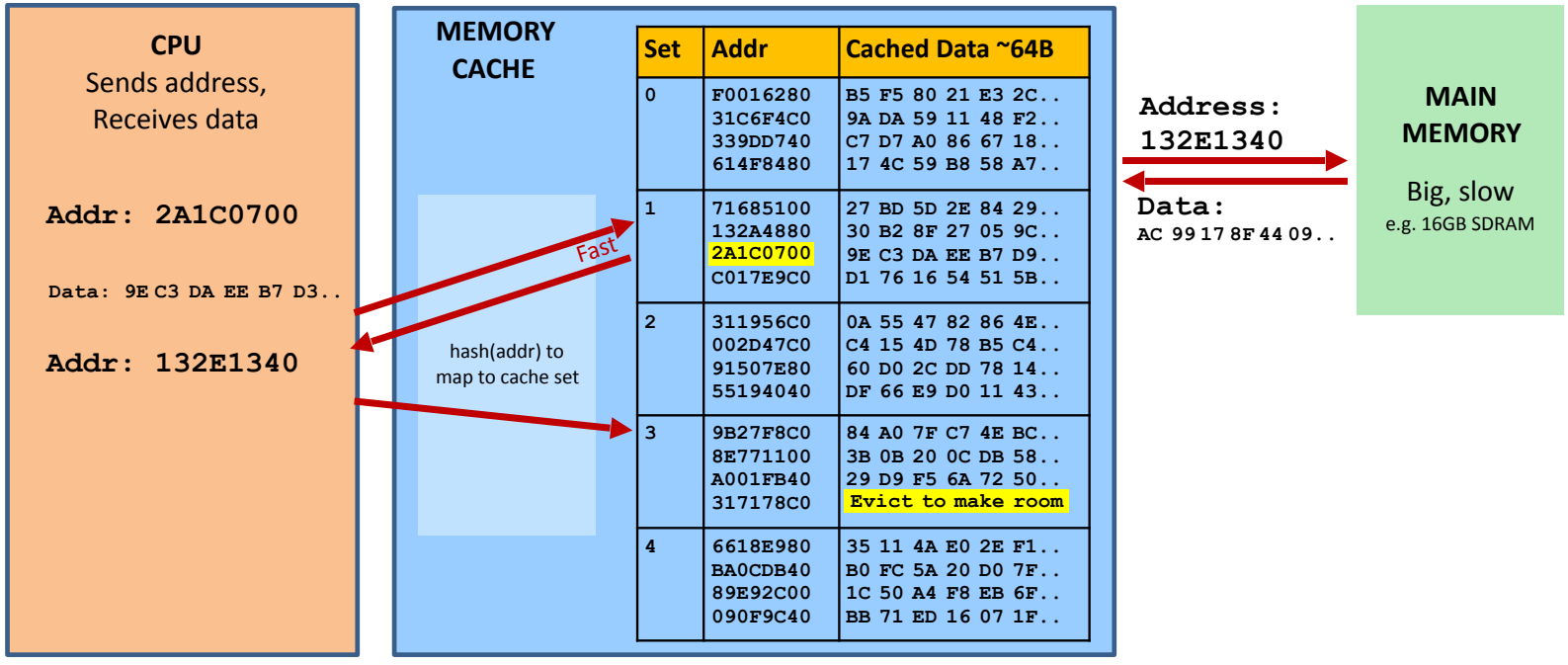
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

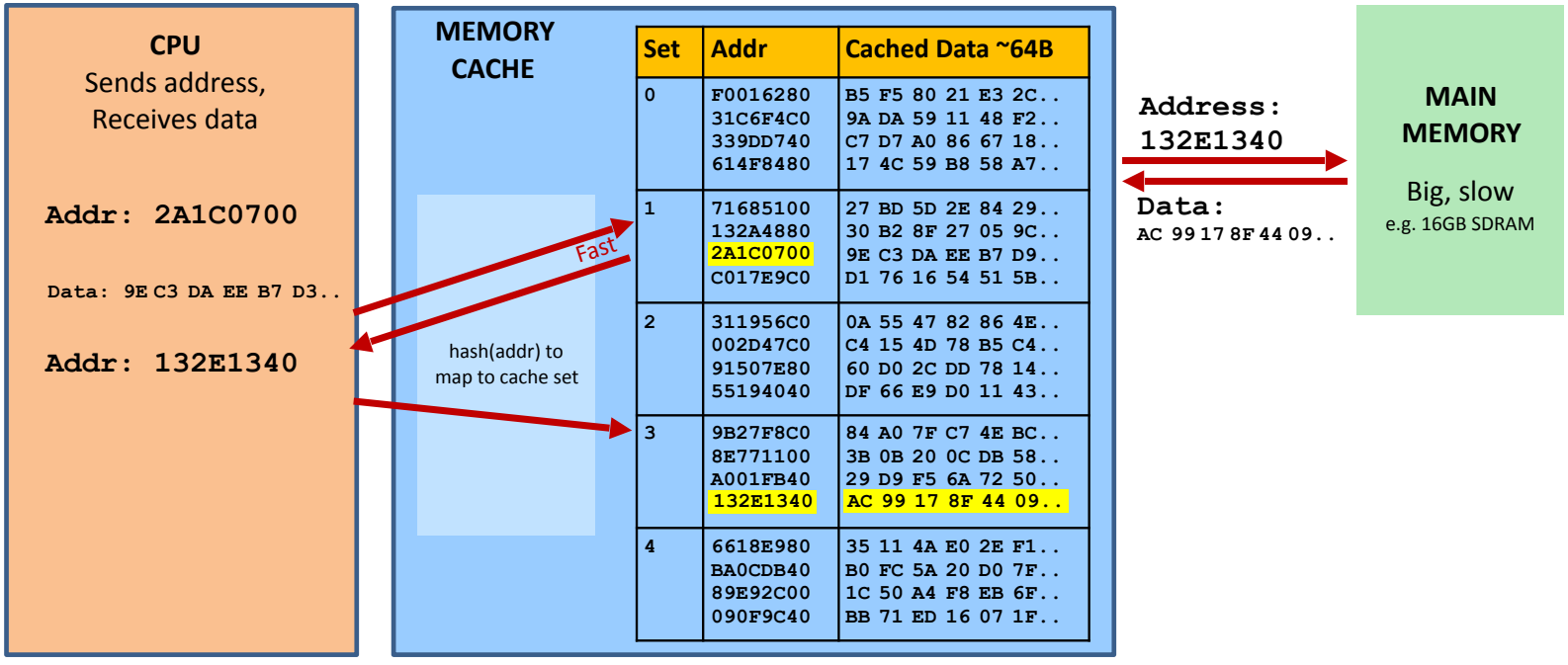
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

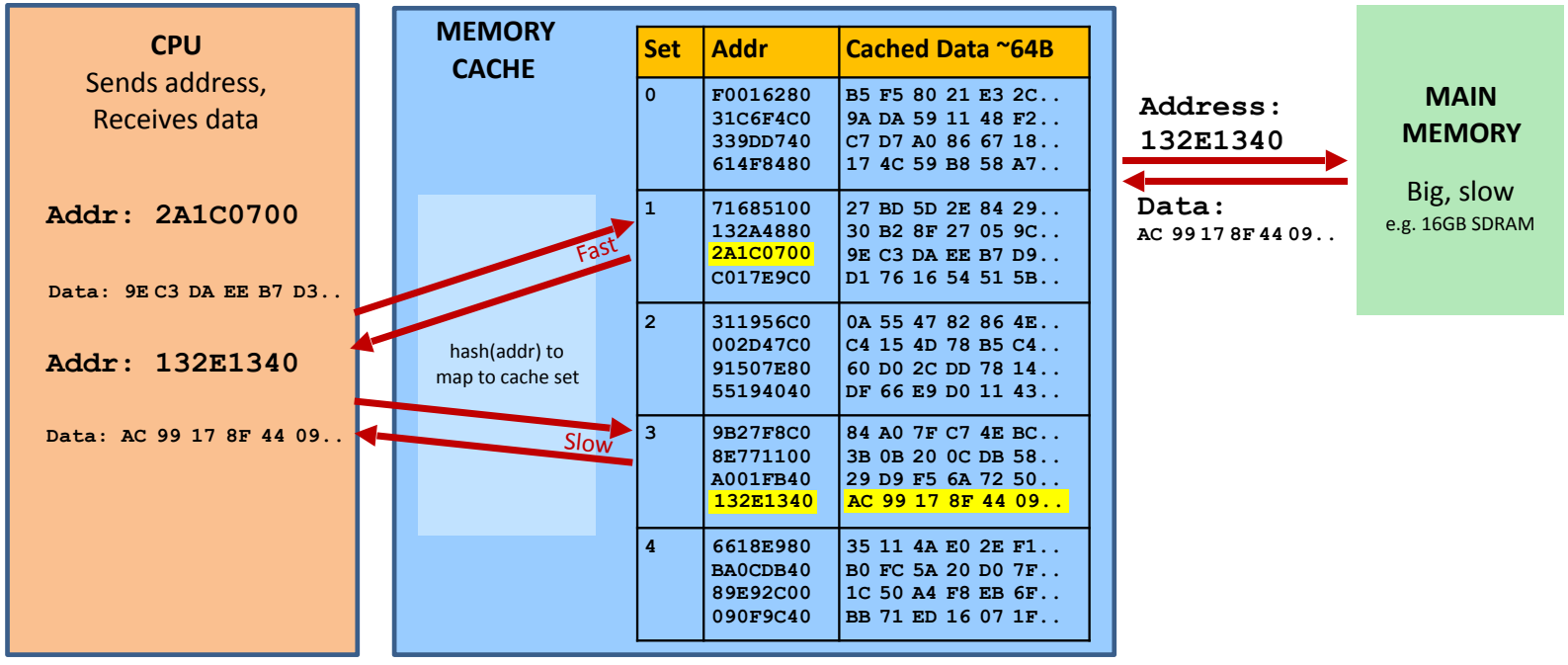
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

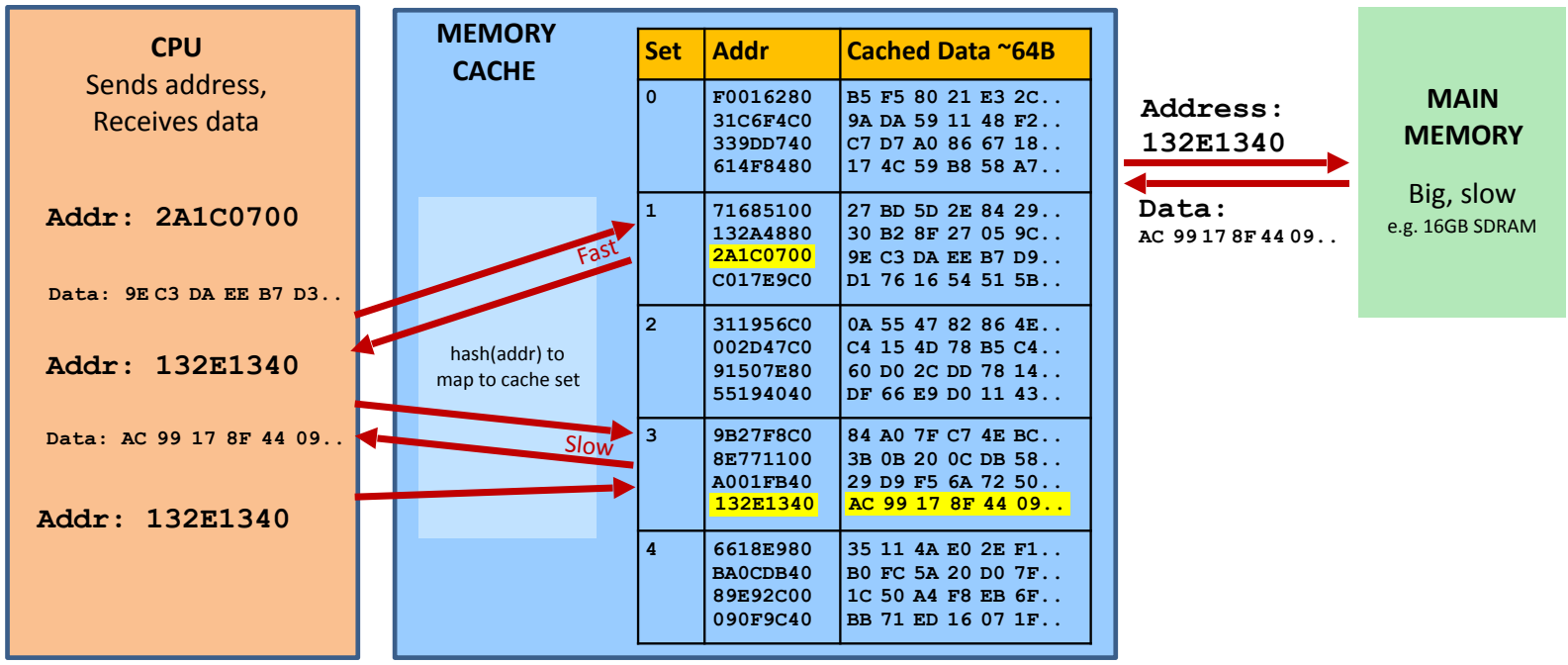
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

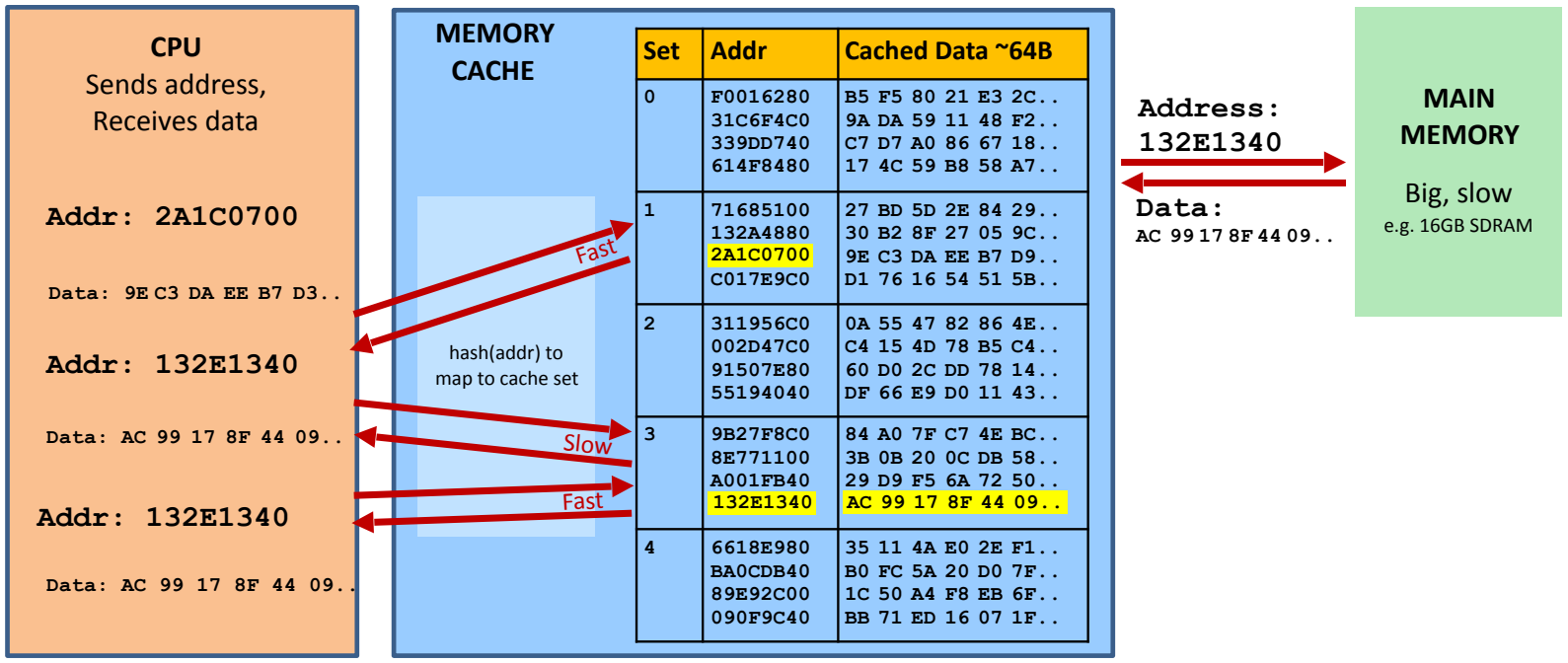
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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

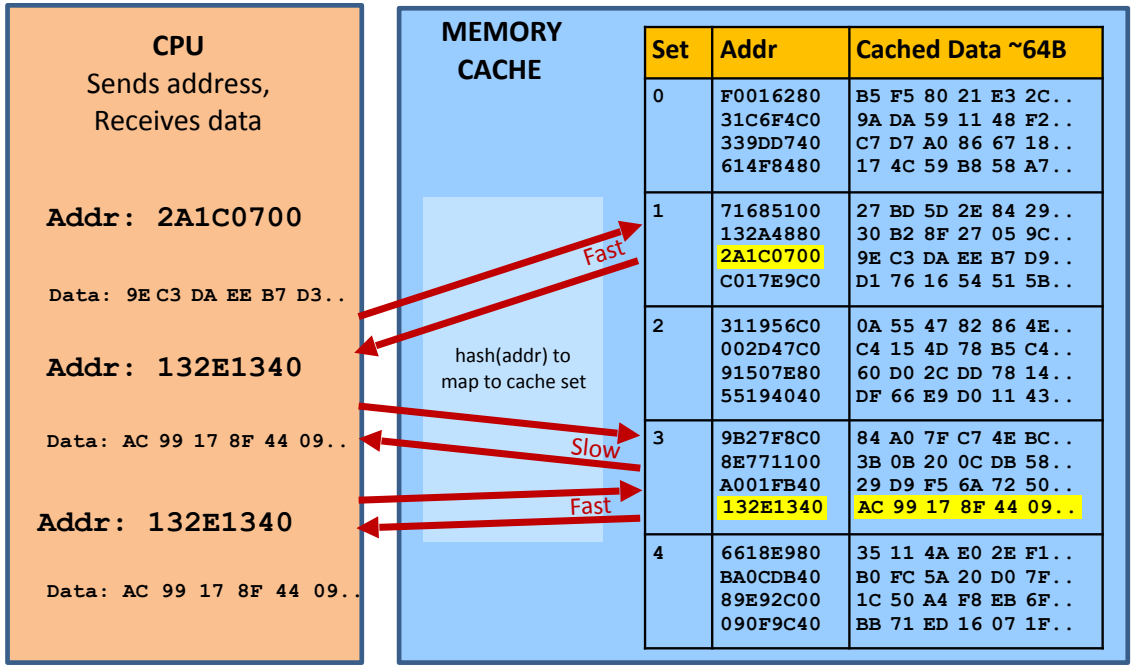
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



- 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



Reads change system state:

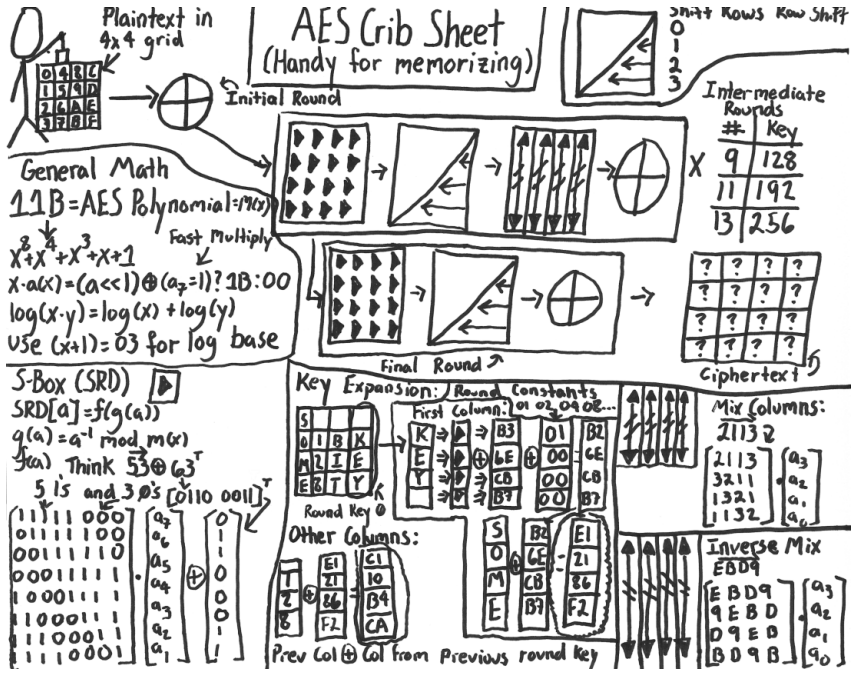
- Read to newly-cached location is fast
- Read to evicted location is slow

- 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 $\rightarrow$ performance gain
  - Inocrrect: Discard speculative work $\rightarrow$ 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

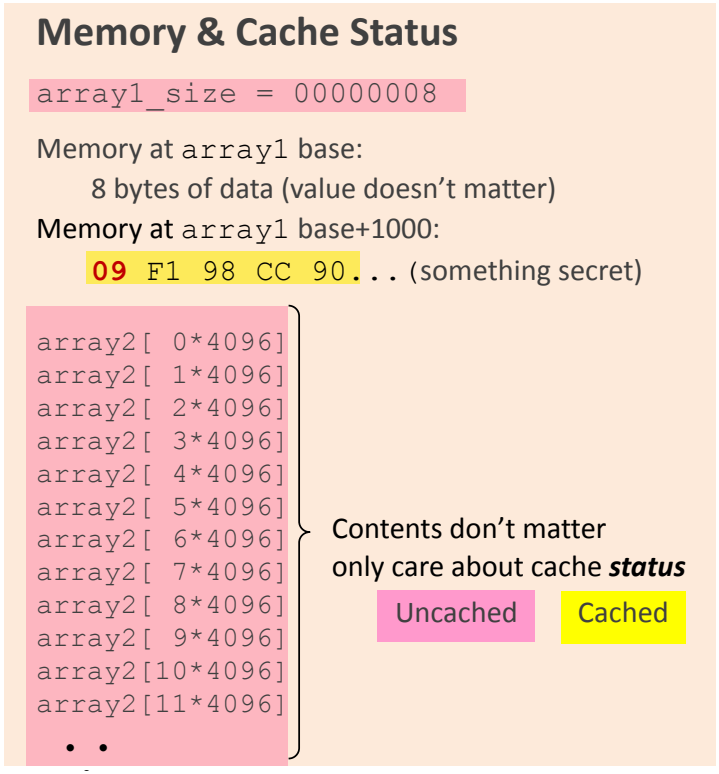
- Spectre and Meltdown carry out cache attacks against speculatively loaded data so that an unprivileged attacker process can read kernel memory, break ASLR, etc.

Spectre variant 1 attack

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

Before attack:

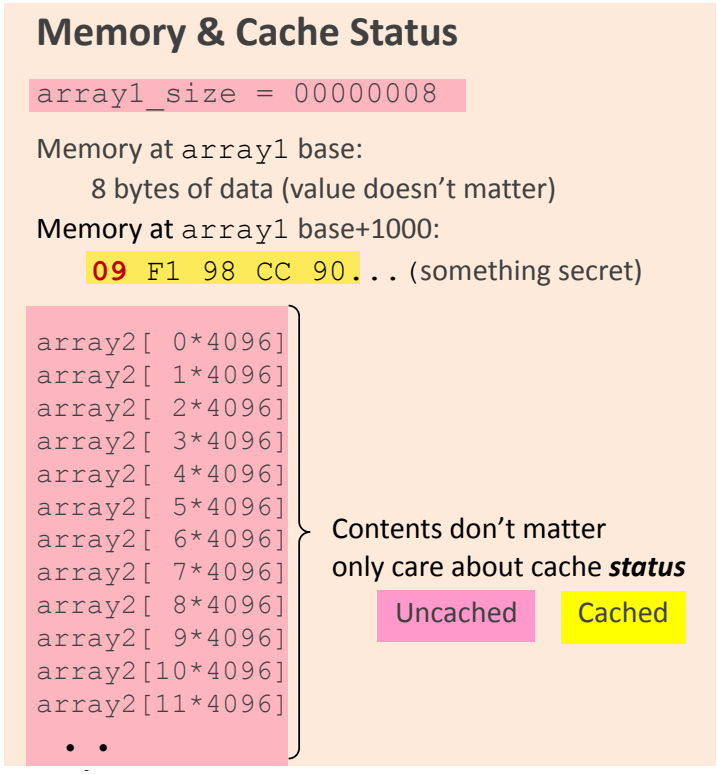
- Train branch predictor to expect if() is true (e.g. call with `x` < `array1_size`)
- Evict `array1_size` and `array2[]` from cache



Spectre variant 1 attack

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

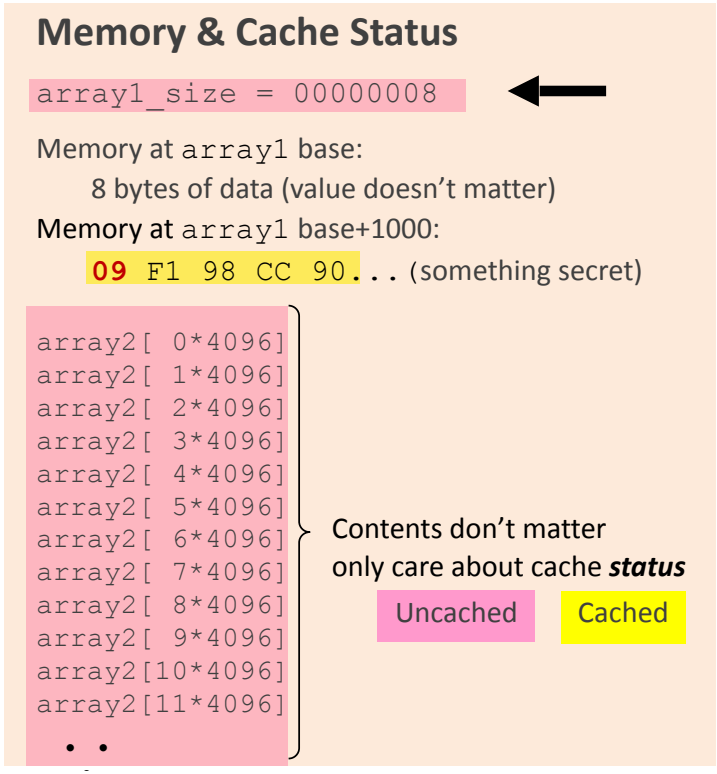
Attacker calls victim with `x`=1000



Spectre variant 1 attack

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

Attacker calls victim with `x`=1000

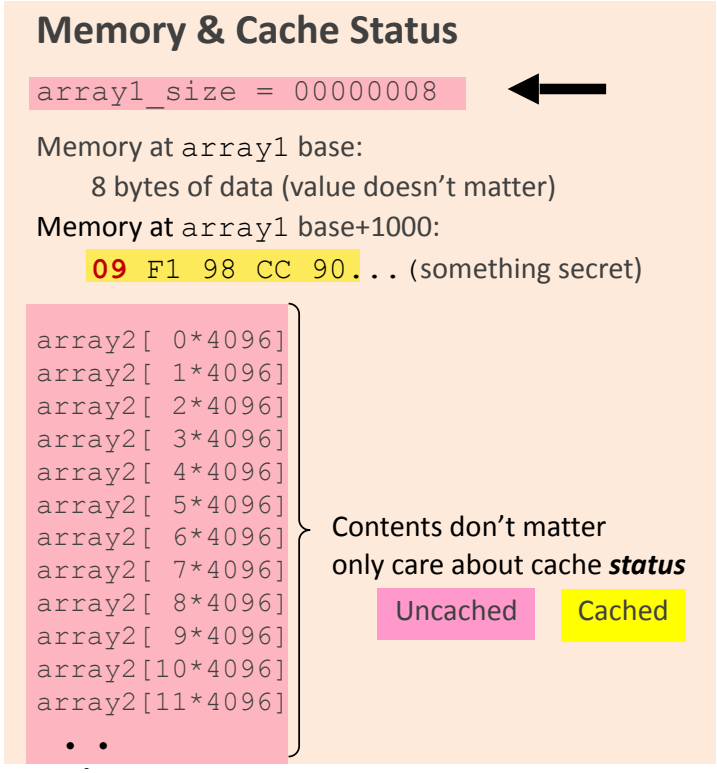


Spectre variant 1 attack

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

Attacker calls victim with `x`=1000

Speculative exec while waiting for `array1_size`:



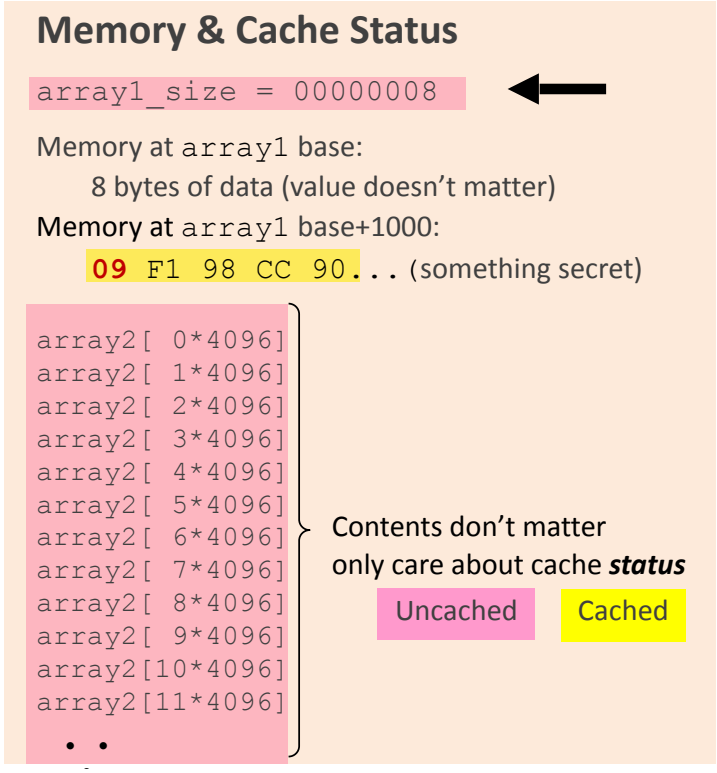
Spectre variant 1 attack

```
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



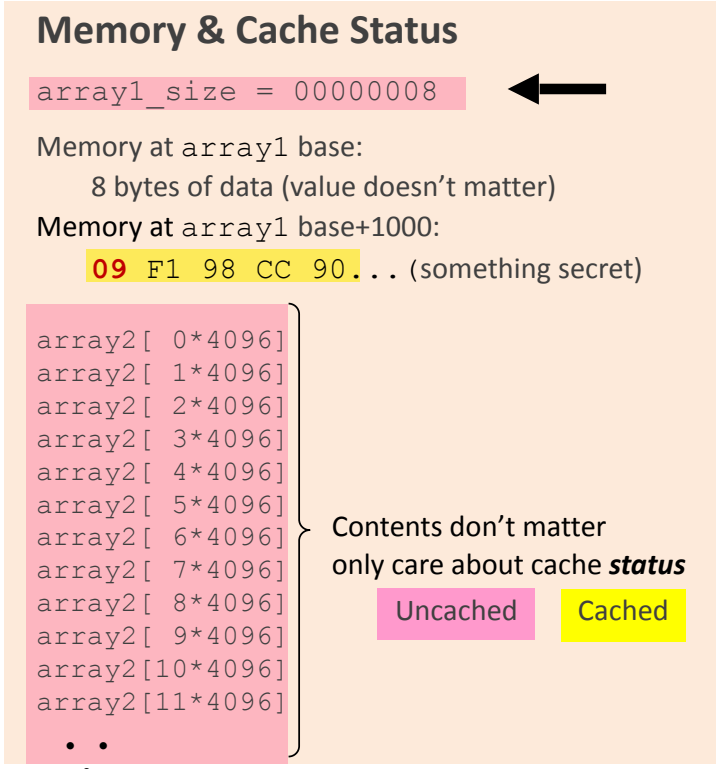
Spectre variant 1 attack

```
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)



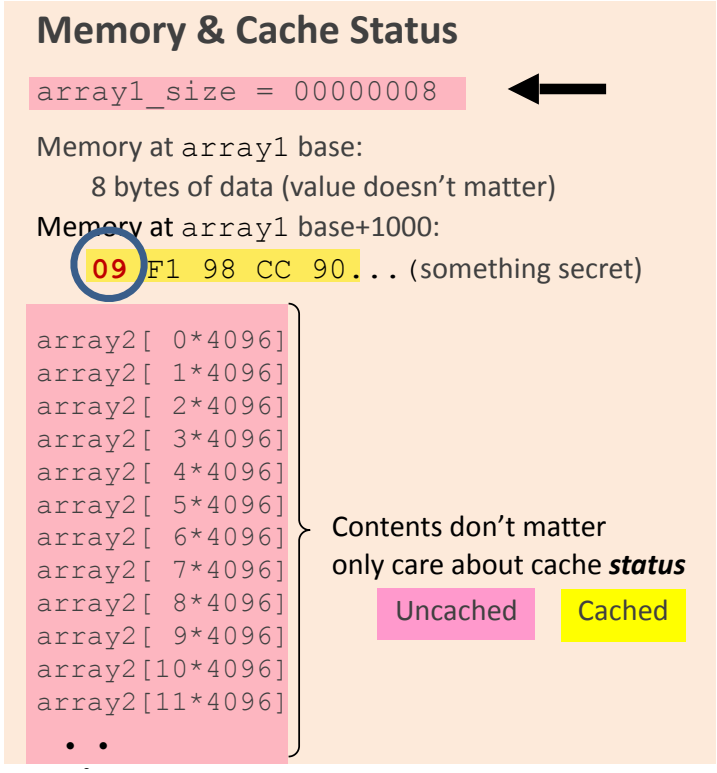
Spectre variant 1 attack

```
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)



Spectre variant 1 attack

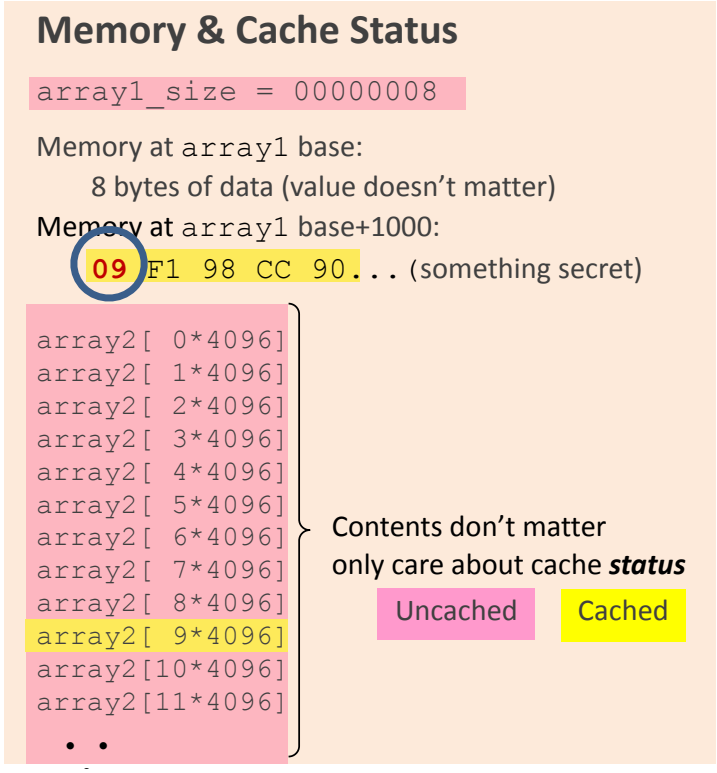
```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

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



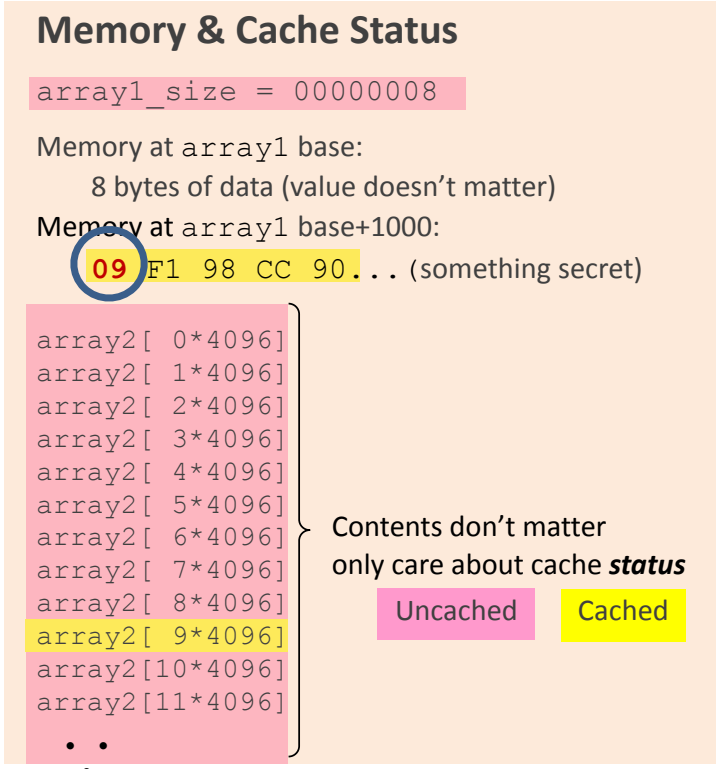
Spectre variant 1 attack

```
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)



Fault Attacks

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

Java heap overflow via glitched address of function pointer.

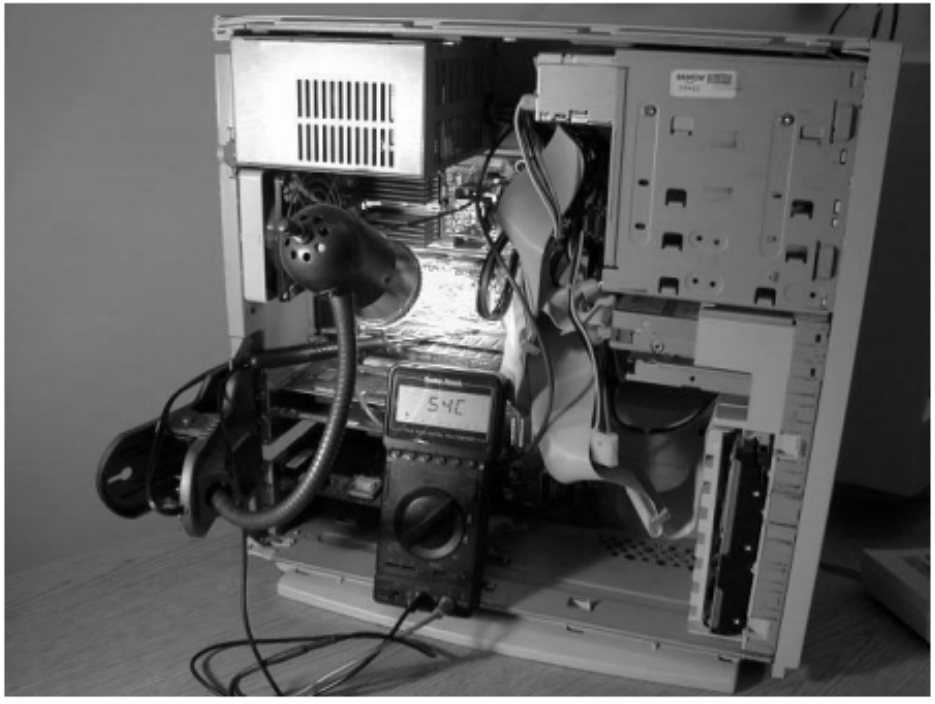


Figure 3. Experimental setup to induce memory errors, showing a PC built from surplus components, clip-on gooseneck lamp, 50-watt spotlight bulb, and digital thermometer. Not shown is the variable AC power supply for the lamp.

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”

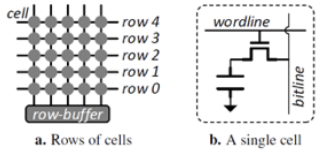
DRAM

- 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!

Code from D. Kohlbrenner

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!

Code from D. Kohlbrenner

Why? Floating-point time variability

Processor	$+$ subnormal	$+$ special	$\times$ subnormal	$\times$ special	$\div$ subnormal	$\div$ special	$\div x^2$	$\div x^4$	$\sqrt{\text{subnormal}}$	$\sqrt{\text{special}}$	$\sqrt{x^2}$	$\sqrt{x^4}$	$\sqrt{-x}$
						Single-precision operations							
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	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗
						Double-precision operations							
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	The architecture makes no statement about the timing properties of any instructions.
0b1	The architecture requires that: • 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: ◦ The values of the data supplied in any of its registers. ◦ The values of the NZCV flags. • For certain data processing instructions, the response of the instruction to asynchronous exceptions does not vary based on: ◦ 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:

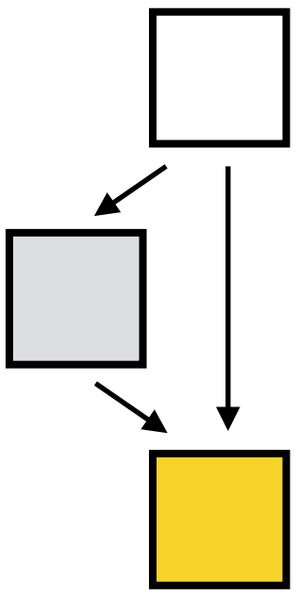
- All cryptographic instructions. These instructions are:
  - AESD, AESE, AESIMC, AESMC, SHA1C, SHA1H, SHA1M, SHA1P, SHA1SU0, SHA1SU1, SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
- A subset of those instructions which use the general-purpose register file. For these instructions, the effects of CPSR.DIT 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?**

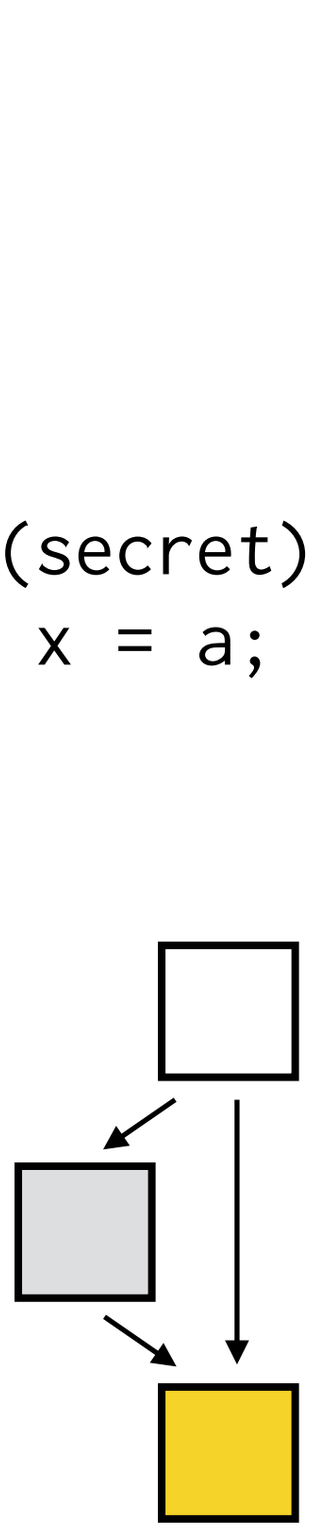
Don’t branch on secrets!

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;
```

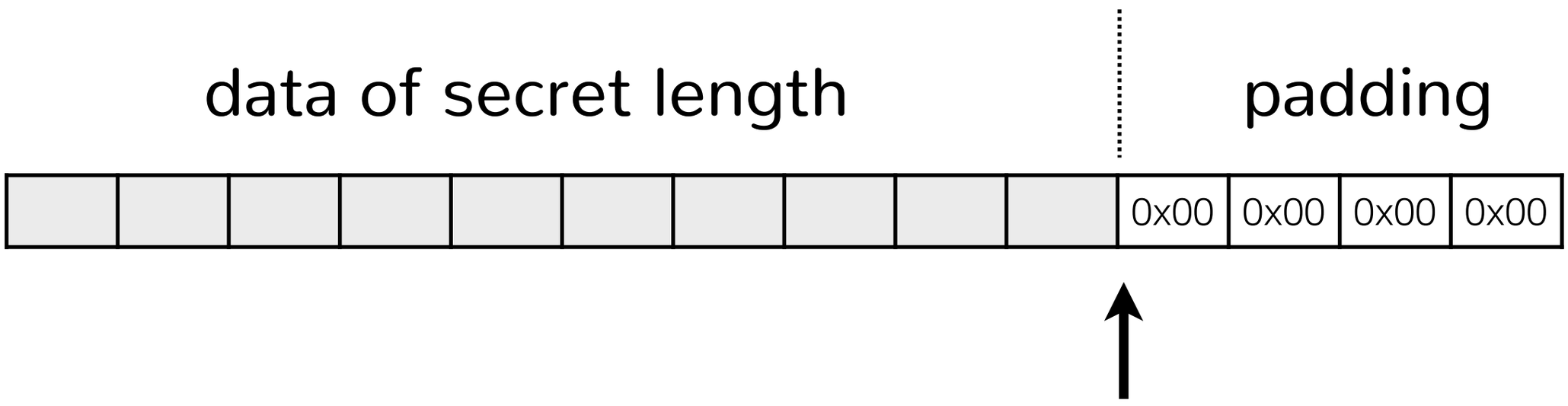
```
x = (1-secret) * b
  + secret * x;
```

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!

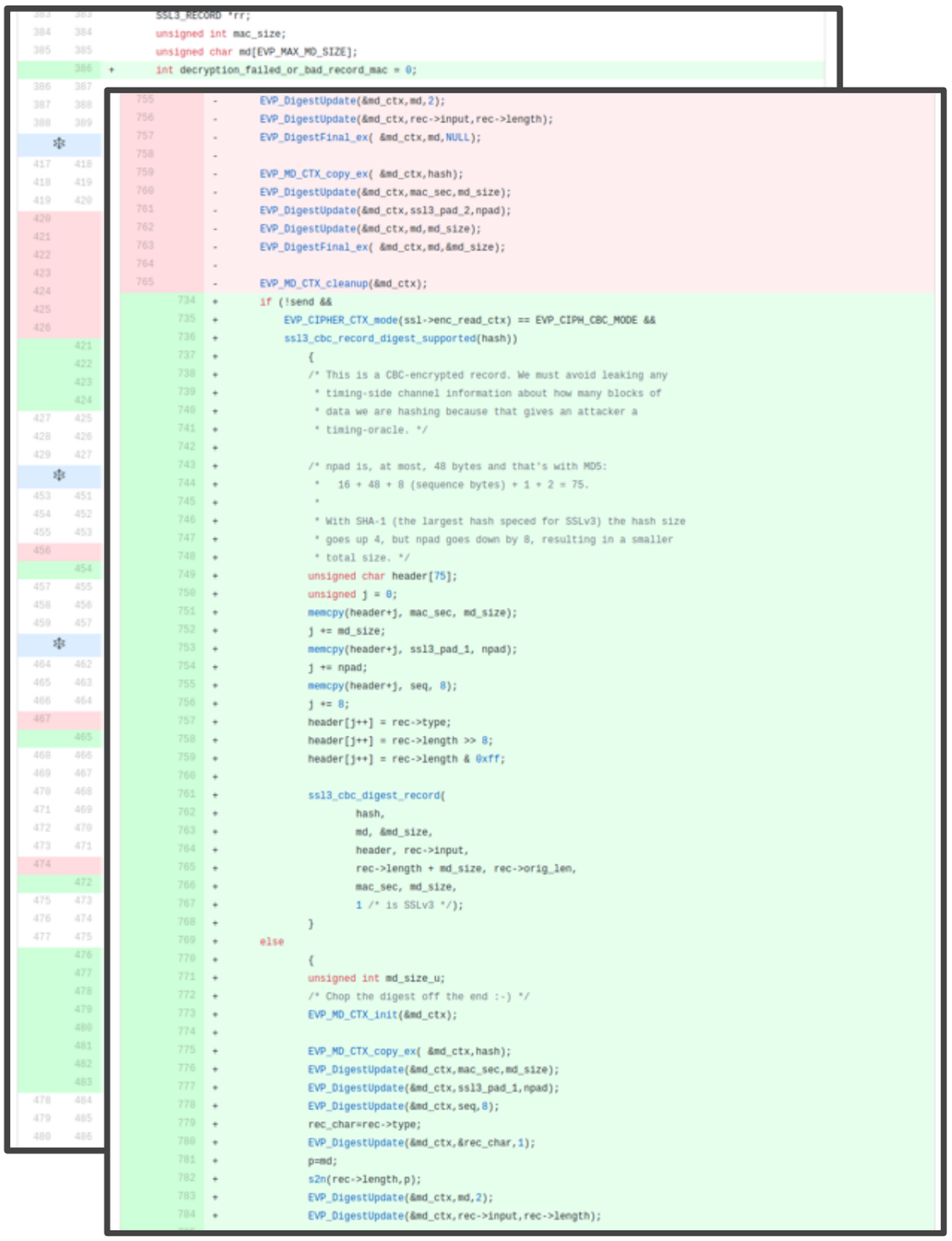
Writing CT code is unholy

```
@@ -384,7 +384,8 @@ dtls1_process_record(SSL *s)
        SSL3_RECORD *rr;
        unsigned int mac_size;
        unsigned char md[EVP_MAX_MD_SIZE];
+       int decryption_failed_or_bad_record_mac = 0;

        rr= &(s->s3->rrec);
@@ -417,13 +418,10 @@ dtls1_process_record(SSL *s)
        enc_err = s->method->ssl3_enc->enc(s,0);
        if (enc_err <= 0)
                {
-               /* decryption failed, silently discard message */
-               if (enc_err < 0)
-                       {
-                       rr->length = 0;
-                       s->packet_length = 0;
-                       }
-               goto err;
+               /* To minimize information leaked via timing, we will always
+                * perform all computations before discarding the message.
+                */
+               decryption_failed_or_bad_record_mac = 1;
                }

 #ifdef TLS_DEBUG
@@ -453,7 +451,7 @@ printf("\n");
                        SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_PRE_MAC_LENGTH_TOO_LONG);
                        goto f_err;
 #else
-                       goto err;
+                       decryption_failed_or_bad_record_mac = 1;
 #endif
                        }
                /* check the MAC for rr->input (it's in mac_size bytes at the tail) */
@@ -464,17 +462,25 @@ printf("\n");
                        SSLerr(SSL_F_DTLS1_PROCESS_RECORD,SSL_R_LENGTH_TOO_SHORT);
                        goto f_err;
 #else
-                       goto err;
+                       decryption_failed_or_bad_record_mac = 1;
 #endif
                        }
                rr->length-=mac_size;
                i=s->method->ssl3_enc->mac(s,md,0);
                if (i < 0 || memcmp(md,&(rr->data[rr->length]),mac_size) != 0)
                        {
-                       goto err;
+                       decryption_failed_or_bad_record_mac = 1;
                        }
                }

+       if (decryption_failed_or_bad_record_mac)
+               {
+               /* decryption failed, silently discard message */
+               rr->length = 0;
+               s->packet_length = 0;
+               goto err;
+               }
+
        /* r->length is now just compressed */
        if (s->expand != NULL)
                {
```

OpenSSL padding oracle attack

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

Writing CT code is unholy



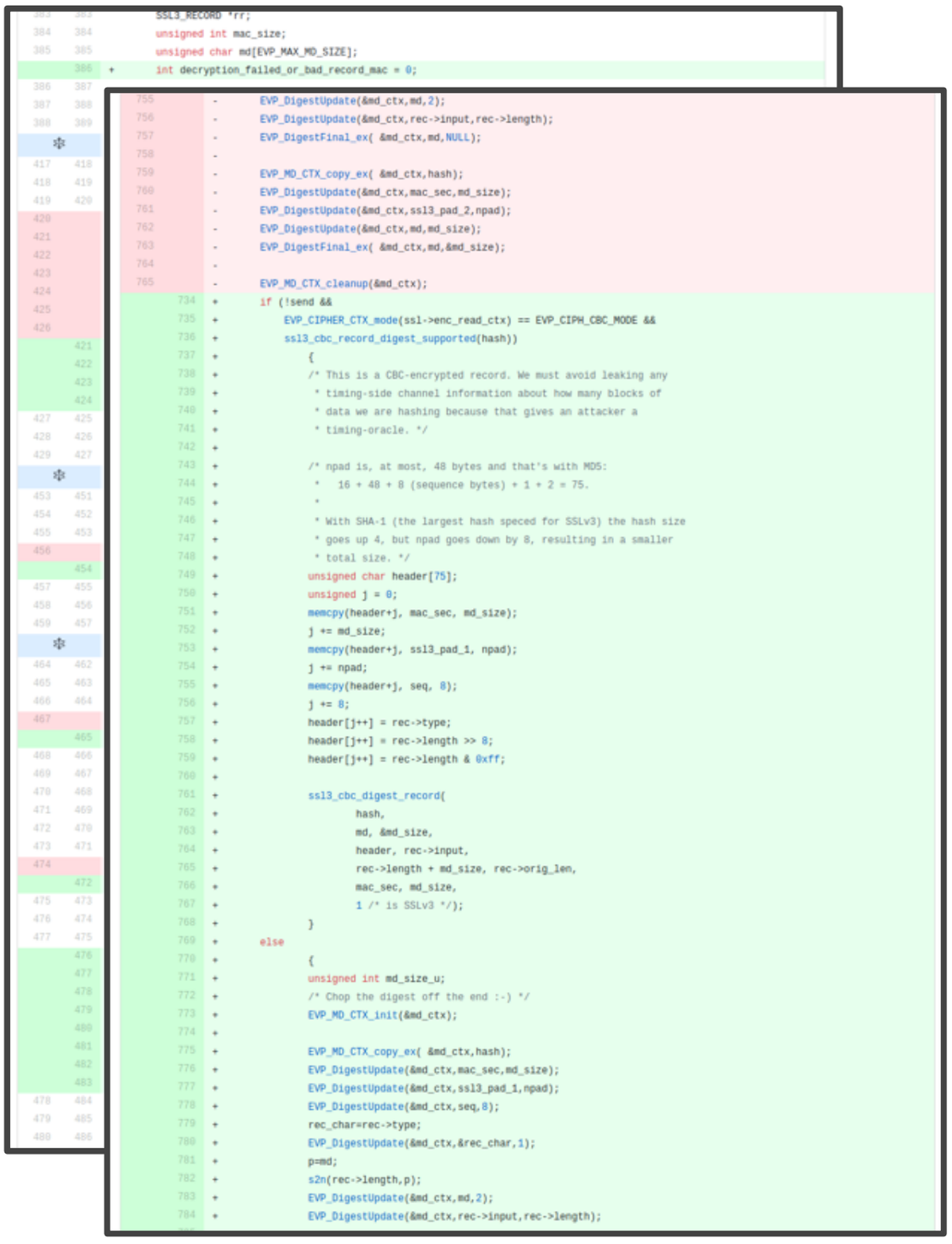
```
SSL3_RECORD *rr;
unsigned int mac_size;
unsigned char md[EVP_MAX_MD_SIZE];
int decryption_failed_or_bad_record_mac = 0;
```

```
-       EVP_DigestUpdate(&md_ctx,md,2);
-       EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
-       EVP_DigestFinal_ex( &md_ctx,md,NULL);
-
-       EVP_MD_CTX_copy_ex( &md_ctx,hash);
-       EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
-       EVP_DigestUpdate(&md_ctx,ssl3_pad_2,npad);
-       EVP_DigestUpdate(&md_ctx,md,md_size);
-       EVP_DigestFinal_ex( &md_ctx,md,&md_size);
-
-       EVP_MD_CTX_cleanup(&md_ctx);
+       if (!send &&
+           EVP_CIPHER_CTX_mode(ssl->enc_read_ctx) == EVP_CIPH_CBC_MODE &&
+           ssl3_cbc_record_digest_supported(hash))
+               {
+               /* This is a CBC-encrypted record. We must avoid leaking any
+                * timing-side channel information about how many blocks of
+                * data we are hashing because that gives an attacker a
+                * timing-oracle. */
+
+               /* npad is, at most, 48 bytes and that's with MD5:
+                *   16 + 48 + 8 (sequence bytes) + 1 + 2 = 75.
+                *
+                * With SHA-1 (the largest hash speced for SSLv3) the hash size
+                * goes up 4, but npad goes down by 8, resulting in a smaller
+                * total size. */
+               unsigned char header[75];
+               unsigned j = 0;
+               memcpy(header+j, mac_sec, md_size);
+               j += md_size;
+               memcpy(header+j, ssl3_pad_1, npad);
+               j += npad;
+               memcpy(header+j, seq, 8);
+               j += 8;
+               header[j++] = rec->type;
+               header[j++] = rec->length >> 8;
+               header[j++] = rec->length & 0xff;
+
+               ssl3_cbc_digest_record(
+                       hash,
+                       md, &md_size,
+                       header, rec->input,
+                       rec->length + md_size, rec->orig_len,
+                       mac_sec, md_size,
+                       1 /* is SSLv3 */);
+               }
+       else
+               {
+               unsigned int md_size_u;
+               /* Chop the digest off the end :-) */
+               EVP_MD_CTX_init(&md_ctx);
+
+               EVP_MD_CTX_copy_ex( &md_ctx,hash);
+               EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
+               EVP_DigestUpdate(&md_ctx,ssl3_pad_1,npad);
+               EVP_DigestUpdate(&md_ctx,seq,8);
+               rec_char=rec->type;
+               EVP_DigestUpdate(&md_ctx,&rec_char,1);
+               p=md;
+               s2n(rec->length,p);
+               EVP_DigestUpdate(&md_ctx,md,2);
+               EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
```

OpenSSL padding oracle attack

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

Writing CT code is unholy



```
SSL3_RECORD *rr;
unsigned int mac_size;
unsigned char md[EVP_MAX_MD_SIZE];
int decryption_failed_or_bad_record_mac = 0;
```

```
-       EVP_DigestUpdate(&md_ctx,md,2);
-       EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
-       EVP_DigestFinal_ex( &md_ctx,md,NULL);
-
-       EVP_MD_CTX_copy_ex( &md_ctx,hash);
-       EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
-       EVP_DigestUpdate(&md_ctx,ssl3_pad_2,npad);
-       EVP_DigestUpdate(&md_ctx,md,md_size);
-       EVP_DigestFinal_ex( &md_ctx,md,&md_size);
-
-       EVP_MD_CTX_cleanup(&md_ctx);
+       if (!send &&
+           EVP_CIPHER_CTX_mode(ssl->enc_read_ctx) == EVP_CIPH_CBC_MODE &&
+           ssl3_cbc_record_digest_supported(hash))
+               {
+               /* This is a CBC-encrypted record. We must avoid leaking any
+                * timing-side channel information about how many blocks of
+                * data we are hashing because that gives an attacker a
+                * timing-oracle. */
+
+               /* npad is, at most, 48 bytes and that's with MD5:
+                *   16 + 48 + 8 (sequence bytes) + 1 + 2 = 75.
+                *
+                * With SHA-1 (the largest hash speced for SSLv3) the hash size
+                * goes up 4, but npad goes down by 8, resulting in a smaller
+                * total size. */
+               unsigned char header[75];
+               unsigned j = 0;
+               memcpy(header+j, mac_sec, md_size);
+               j += md_size;
+               memcpy(header+j, ssl3_pad_1, npad);
+               j += npad;
+               memcpy(header+j, seq, 8);
+               j += 8;
+               header[j++] = rec->type;
+               header[j++] = rec->length >> 8;
+               header[j++] = rec->length & 0xff;
+
+               ssl3_cbc_digest_record(
+                       hash,
+                       md, &md_size,
+                       header, rec->input,
+                       rec->length + md_size, rec->orig_len,
+                       mac_sec, md_size,
+                       1 /* is SSLv3 */);
+               }
+       else
+               {
+               unsigned int md_size_u;
+               /* Chop the digest off the end :-) */
+               EVP_MD_CTX_init(&md_ctx);
+
+               EVP_MD_CTX_copy_ex( &md_ctx,hash);
+               EVP_DigestUpdate(&md_ctx,mac_sec,md_size);
+               EVP_DigestUpdate(&md_ctx,ssl3_pad_1,npad);
+               EVP_DigestUpdate(&md_ctx,seq,8);
+               rec_char=rec->type;
+               EVP_DigestUpdate(&md_ctx,&rec_char,1);
+               p=md;
+               s2n(rec->length,p);
+               EVP_DigestUpdate(&md_ctx,md,2);
+               EVP_DigestUpdate(&md_ctx,rec->input,rec->length);
```

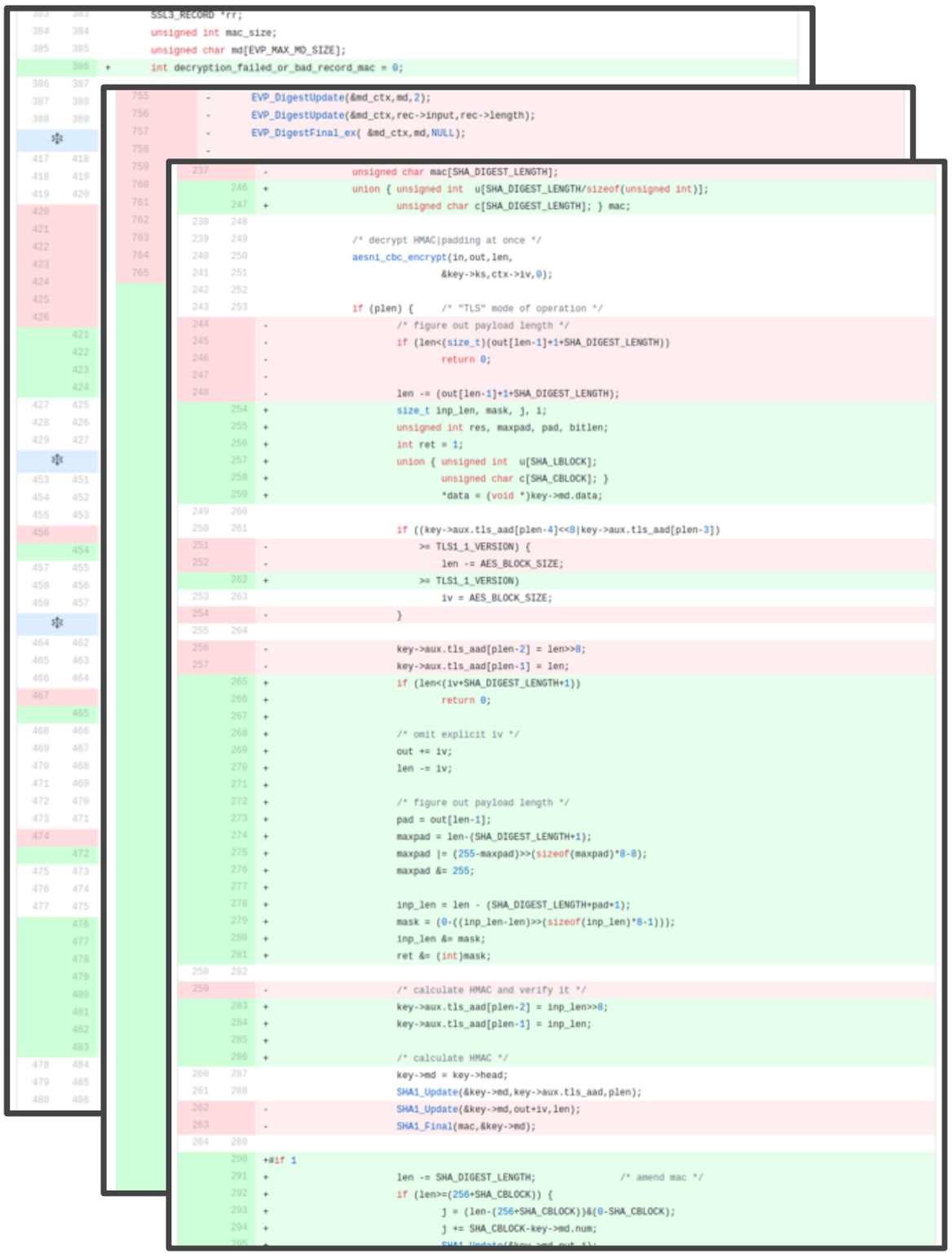
OpenSSL padding oracle attack

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



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

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

```
583  584          maxpad |= (255 - maxpad) >> (sizeof(maxpad) * 8 - 8);
584  585          maxpad &= 255;
585  586
     587  +       ret &= constant_time_ge(maxpad, pad);
     588  +
586  589          inp_len = len - (SHA_DIGEST_LENGTH + pad + 1);
587  590          mask = (0 - ((inp_len - len) >> (sizeof(inp_len) * 8 - 1)));
588  591          inp_len &= mask;
```

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.

CVE-2016-2107

Somorovsky. “Curious padding oracle in OpenSSL.”

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?

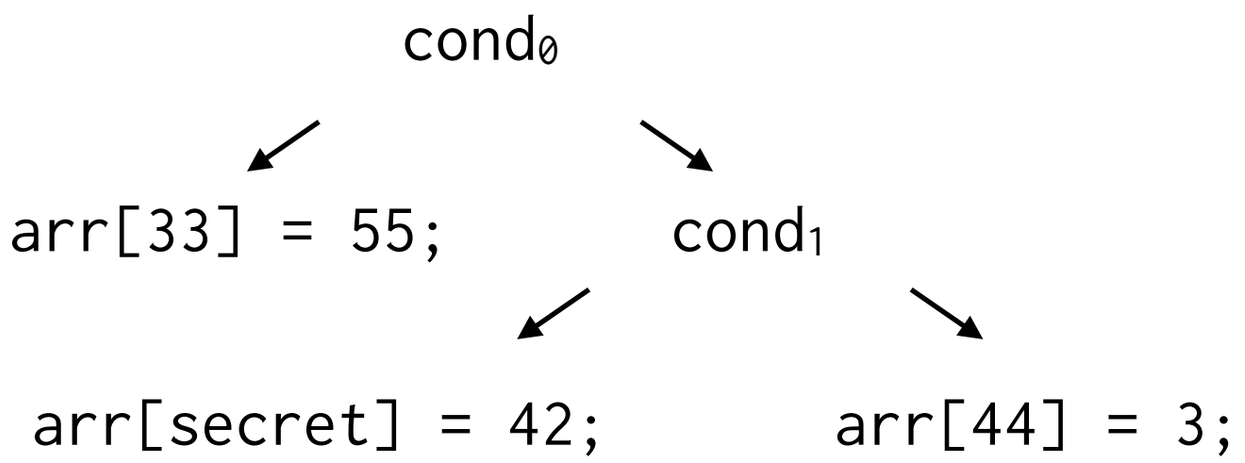
- Program analysis to the rescue!
  - Symbolically execute a function and alert if you find an input that causes it to (1) branch on a secret (2) use secret as memory index or (3) use variable-time instruction

```
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

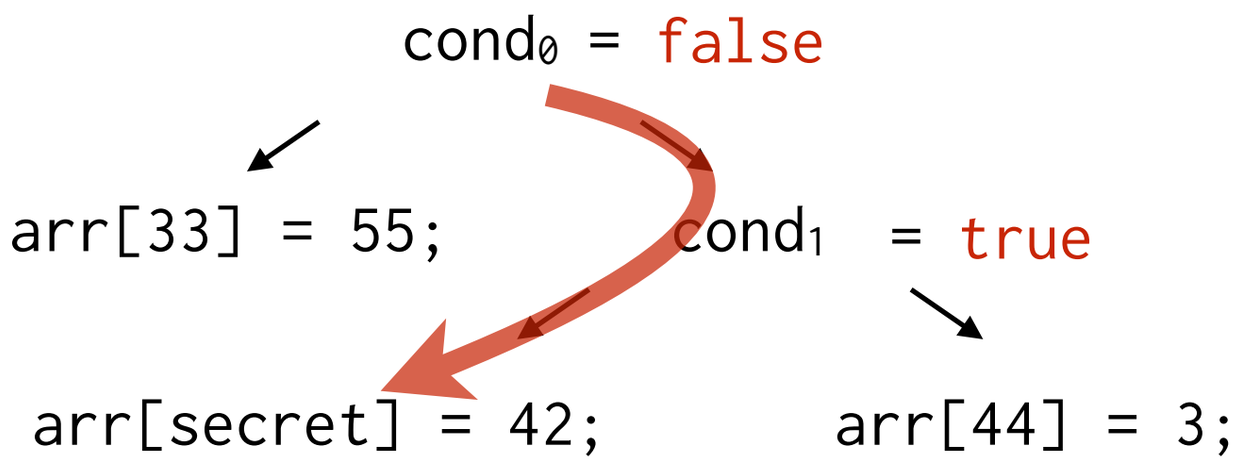
```
if (cond0) {
  arr[33] = 5;
} else {
  if (cond1) {
    arr[secret] = 42;
  } else {
    arr[44] = 3;
 }
}
```



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!