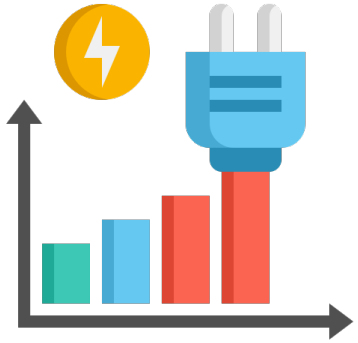
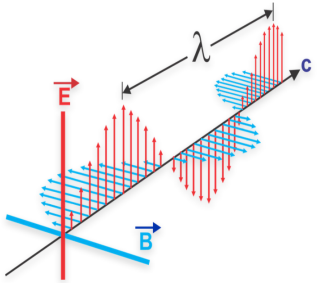
Side Channels

A **side channel** is visible information about how your program runs that is not part of its normal input and output

timing



power



emissions

remenance

String Comparison

```
int check_password(char *provided, int p_len,
                   char *expected, int e_len) {
  if (p_len != e_len)
    return 0;

  for (int i = 0; i < p_len; i++)
    if (provided[i] != expected[i])
       return 0;

  return 1;
}
```

""	1ns
"a"	1ns
…	
"aaaaa"	1ns
"aaaaaa"	2ns
"baaaaa"	2ns
…	
"saaaaa"	3ns
…	
"secaaa"	5ns
…	
"secret"	success!

Timing-Safe String Comparison

```
int check_password(char *provided, int p_len,
                   char *expected, int e_len) {
  int ok = (p_len == e_len);

  for (int i = 0; i < p_len; i++)
    ok = ok & (i < e_len) & (provided[i] == expected[i % e_len]);

  return ok;
}
```

Time to check depends only on provided string length

Timing-Safe String Comparison

```
int check_password(char *provided, int p_len,
                   char *expected, int e_len) {
  int ok = (p_len == e_len);

  for (int i = 0; i < p_len; i++)
    ok = ok & (i < e_len) & (provided[i] == expected[i % e_len]);

  return ok;
}
```

Time to check depends only on provided string length

... unless the C compiler notices a shortcut

Cauligi et al., SecDev 2017

FaCT: A Flexible, Constant-Time Programming Language

Sunjay Cauligi[†] Gary Soeller[†] Fraser Brown[⋆] Brian Johannesmeyer[†] Yunlu Huang[†]
Ranjit Jhala[†] Deian Stefan[†]
[⋆]Stanford University [†]UC San Diego

***Abstract*—We argue that C is unsuitable for writing timing-channel free cryptographic code that is both fast and readable. Readable implementations of crypto routines would contain high-level constructs like `if` statements, constructs that *also* introduce timing vulnerabilities. To avoid vulnerabilities, programmers must rewrite their code to dodge intuitive yet dangerous constructs, cluttering the codebase and potentially introducing new errors. Moreover, even when programmers are diligent, compiler optimization passes may still introduce branches and other sources of timing side channels. This status quo is the worst of both worlds: tortured source code that can still yield vulnerable machine code. We propose to solve this problem with a domain-specific *language* that permits programmers to intuitively express crypto routines and reason about secret values, and a *compiler* that generates efficient, timing-channel free assembly code.**

been susceptible to timing attacks that allowed (even remote network) attackers to recover secret keys.

To avoid introducing timing vulnerabilities, developers translate unsafe C functions into safe *constant-time* functions using a selection of standard recipes. For example, to implement a safe version of the conditional assignment from the RSA function above, the alert developer rewrites "`if (secret) x = expr`" to avoid branching on the secret; they choose their favorite "recipe" to convert the snippet into constant-time code. Conceptually, this vulnerable code is equivalent to the branchless arithmetic assignment "`x = (secret * expr) + (1 - secret) * x`." In this rewrite, `x` takes on the value `expr` if the `secret` bit is equal to one; it remains the same when `secret` is equal

Another Idea: Don't Compare Password Strings

As you know, a server should not store and compare password strings

Comparing *hashes* is not so dangerous:

- Hash of input depends only on the input length
- Getting a prefix of a hash right doesn't help find the whole hash

Modular Exponentiation for RSA Decryption

$$\text{Dec}(x) = x^{\text{private\_key}} \bmod N$$

```
result = x;
for (int i = KEY_BITS-1; i >= 0; i--) {
   result = (result * result) % N;
   if (is_bit_set(private_key, i))
     result = (x * result) % N;
}
```

+ Iteration count is independent of private key
- More work every time a key bit is set

  ⇒ timing indicates number of bits set
- The extra work is reflected not only by time, but power!

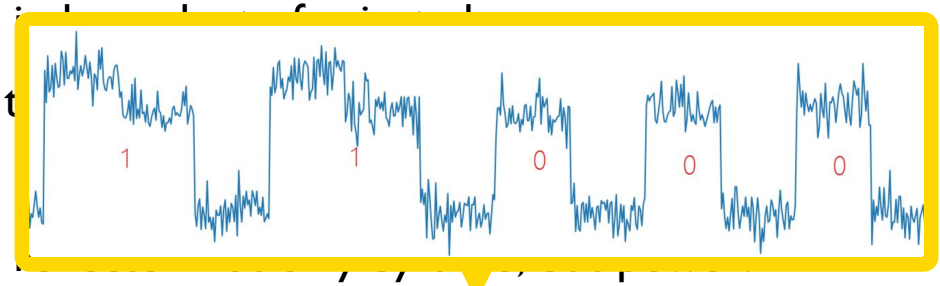
  ⇒ power indicates *which* bits are set

Modular Exponentiation for RSA Decryption

$$Dec(x) = x^{private\_key} \bmod N$$

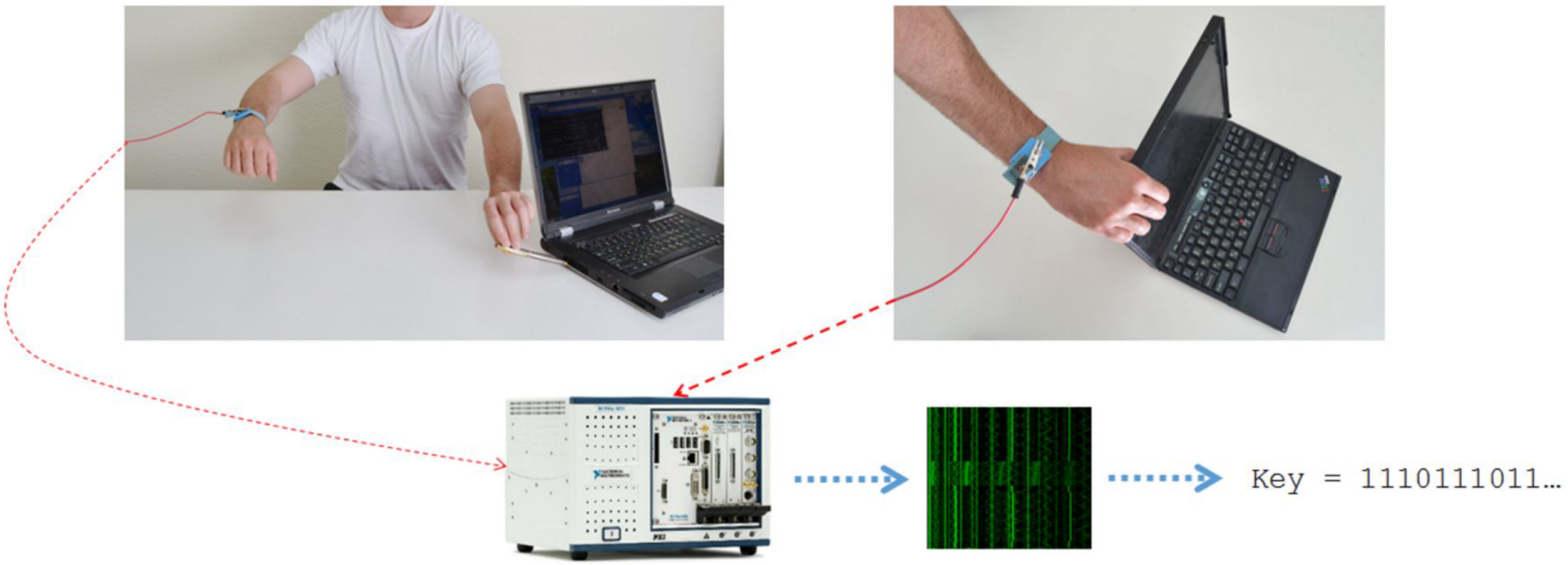
```
result = x;
for (int i = KEY_BITS-1; i >= 0; i--) {
   result = (result * result) % N;
   if (is_bit_set(private_key, i))
     result = (x * result) % N;
}
```

+ Iteration count is i
- More work every t
- The extra work is



⇒ power indicates *which* bits are set

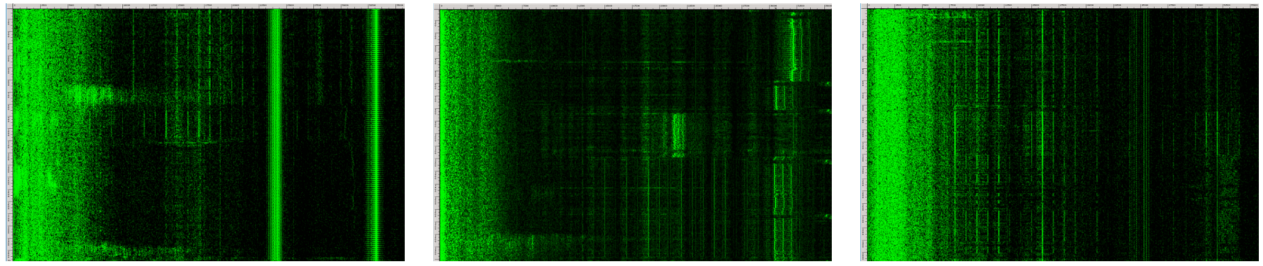
Genkin et al., CHES 2014



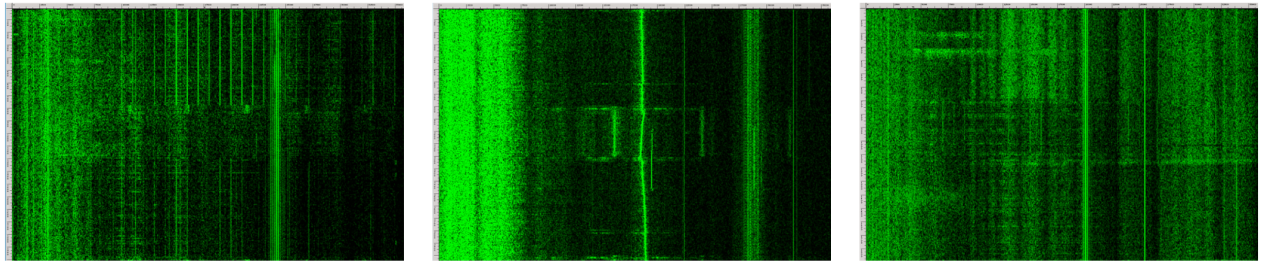
Same idea works with electromagnetic field measurements!

Genkin et al., CRYPTO 2014

... or even with sound!



(a) Asus N55SF (b) Dell Inspiron 7720 (c) HP ProBook 4530s



(d) HP Pavilion Sleekbook 15-b005ej (e) Samsung NP300V5A (f) Lenovo ThinkPad W530

Figure 8: Acoustic emanations (2 sec, 0–35 kHz) of various target computers performing MUL, HLT and MEM in this order. Note that the three operations can be distinguished on all machines.

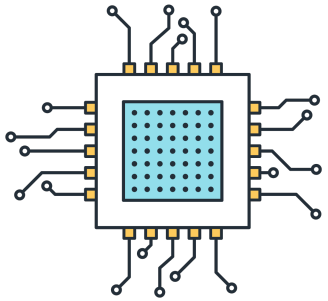
Acoustic Side Channel Attack

Harrison et al. (EuroS&PW'23): practical with 25 samples from each of 36 keys

Differential Fault Analysis

Differential fault analysis pokes at hardware to see what happens:

- apply too-high or too-low voltage
- short pins
- expose to radiation



... well outside anything you have to worry about in software

Remenance

```
void decrypt(....) {
  key private_key;

  load_private_key(&private_key);

  rsa_decrypt(&private_key, ....);
}

void work(....) {
  char buffer[16];
  ....
}

int main() {
  decrypt(....);
  work(....);
}
```

Remenance

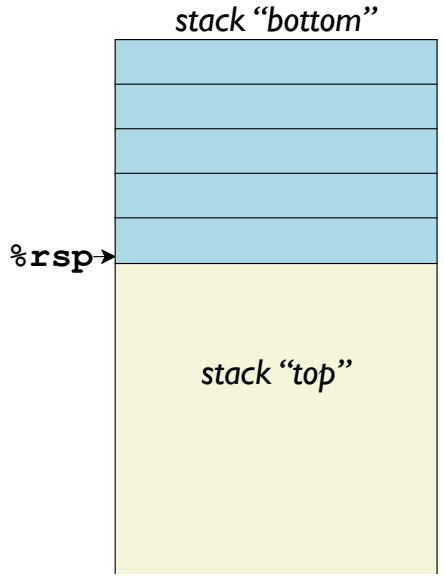
```
void decrypt(....) {
  key private_key;

  load_private_key(&private_key);

  rsa_decrypt(&private_key, ....);
}

void work(....) {
  char buffer[16];
  ....
}

int main() {
  decrypt(....);
  work(....);
}
```



Remenance

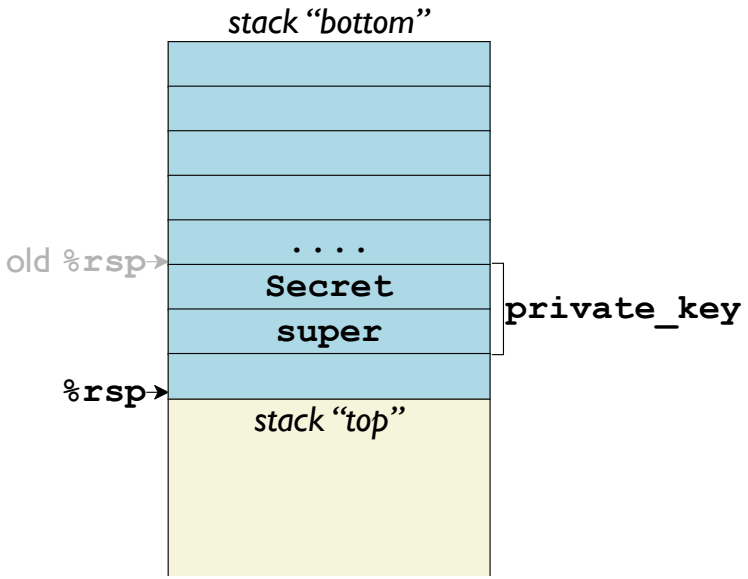
```
void decrypt(....) {
  key private_key;

  load_private_key(&private_key);

  rsa_decrypt(&private_key, ....);
}

void work(....) {
  char buffer[16];
  ....
}

int main() {
  decrypt(....);
  work(....);
}
```



Remenance

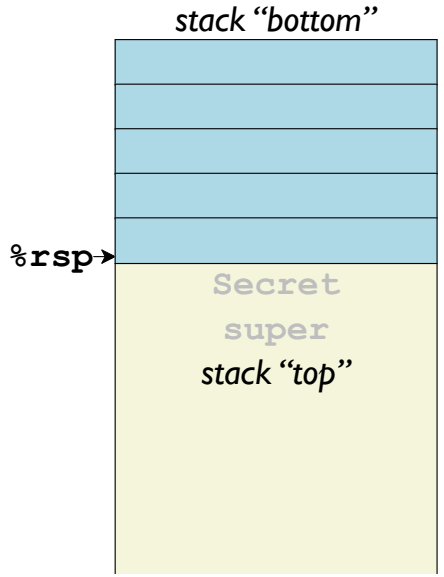
```
void decrypt(....) {
  key private_key;

  load_private_key(&private_key);

  rsa_decrypt(&private_key, ....);
}

void work(....) {
  char buffer[16];
  ....
}

int main() {
  decrypt(....);
  work(....);
}
```



Remenance

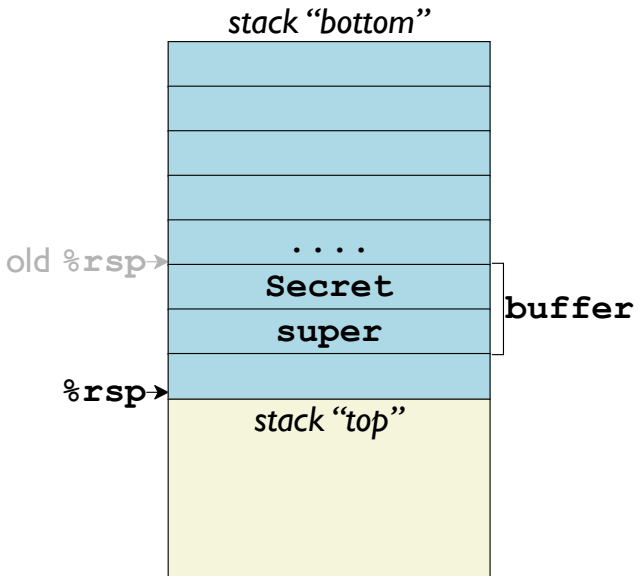
```
void decrypt(....) {
  key private_key;

  load_private_key(&private_key);

  rsa_decrypt(&private_key, ....);
}

void work(....) {
  char buffer[16];
  ....
}

int main() {
  decrypt(....);
  work(....);
}
```



Remenance

Stack-allocated values are not erased on return

Heap-allocated values are not erased by `free`

C compiler is likely to eliminate an unnecessary(!) `memset`

```
#include <string.h>

extern void go(char *);

int work() {
  char buffer[10];
  go(buffer);
  memset(buffer, 0, 10);
  return 0;
}
```

```
_work:
00000000  subq  $0x18, %rsp
00000004  leaq  0xe(%rsp), %rdi
00000009  callq _go
0000000e  xorl  %eax, %eax
00000010  addq  $0x18, %rsp
00000014  retq
```

Remenance

Stack-allocated values are not erased on return

Heap-allocated values are not erased by `free`

C compiler is likely to eliminate an unnecessary(!) `memset`

Java's `SecretKey` class versus `Key`:

- `Key` has no destructor
- `SecretKey` destructor zeroes out memory

Unlike `malloc`, the `mmap` system call delivers zeroed memory

Summary

A **side channel** exposes program behavior outside of normal I/O channels:

timing, power, EM, acoustics, remenance

A **side channel attack** tries to take advantage of side channels

Typical practice: use good languages and APIs, then hope for the best!