Week 13: Lecture A
Side Channels & Hardware Security

Tuesday, November 19, 2024

Announcements

- **Project 3** grades are now available on **Canvas**
- **Statistics:**
  - Average score: **97%**
  - Last year's avg: **90%**
- **Fantastic job!**
- **Regrades coming soon!**

Announcements

- **Project 4: NetSec** released
  - **Deadline:** Thursday, December 5th by 11:59PM

Project 4: Network Security

Deadline: Thursday, December 5 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

Project 4 Progress

Working on Part 1 — 0%

Finished Part 1, working on Part 2 — 0%

Finished both Part 1 and Part 2 — 0%

None of the above — 0%

Final Exam

- **Save the date:** **1–3PM** on **Tuesday, December 10**
  - **CDA accommodations:** schedule exam via CDA Portal

- **High-level details** (more to come):
  - One exam covering all course material
  - Similar to project/quiz/lecture exercises

Final Exam

- **Save the date: 1–3PM** on **Tuesday, December 10**
  - **CDA accommodations:** schedule exam via CDA Portal
- **High-level details** (more to come):
  - One exam covering all course material
  - Similar to project/quiz/lecture exercises
- **Practice Exam** will be released this Thursday
  - See **Assignments** page on the CS 4440 website
- **Final lecture** will serve as a **review session**
  - Practice Exam solutions discussed **in-class only**—don't skip!

Announcements

utahsec

See Discord for meeting info!

utahsec.cs.utah.edu

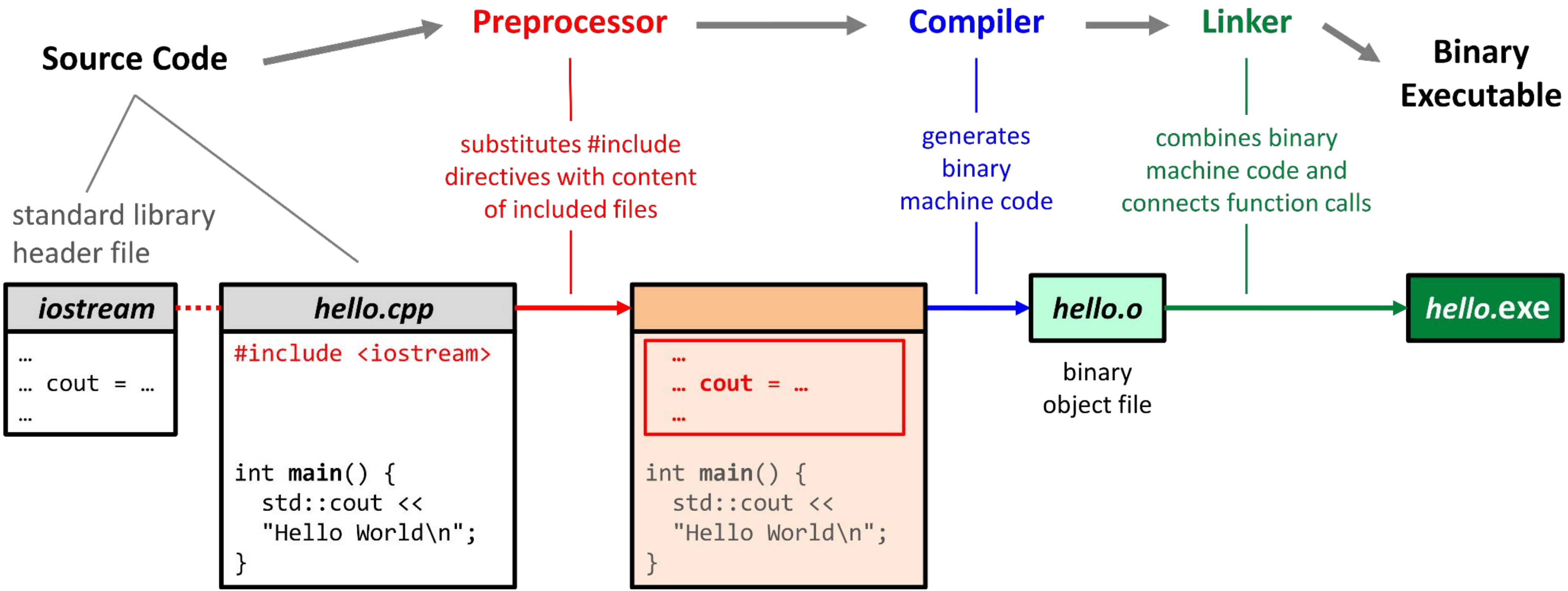
Questions?

No Class Next Week

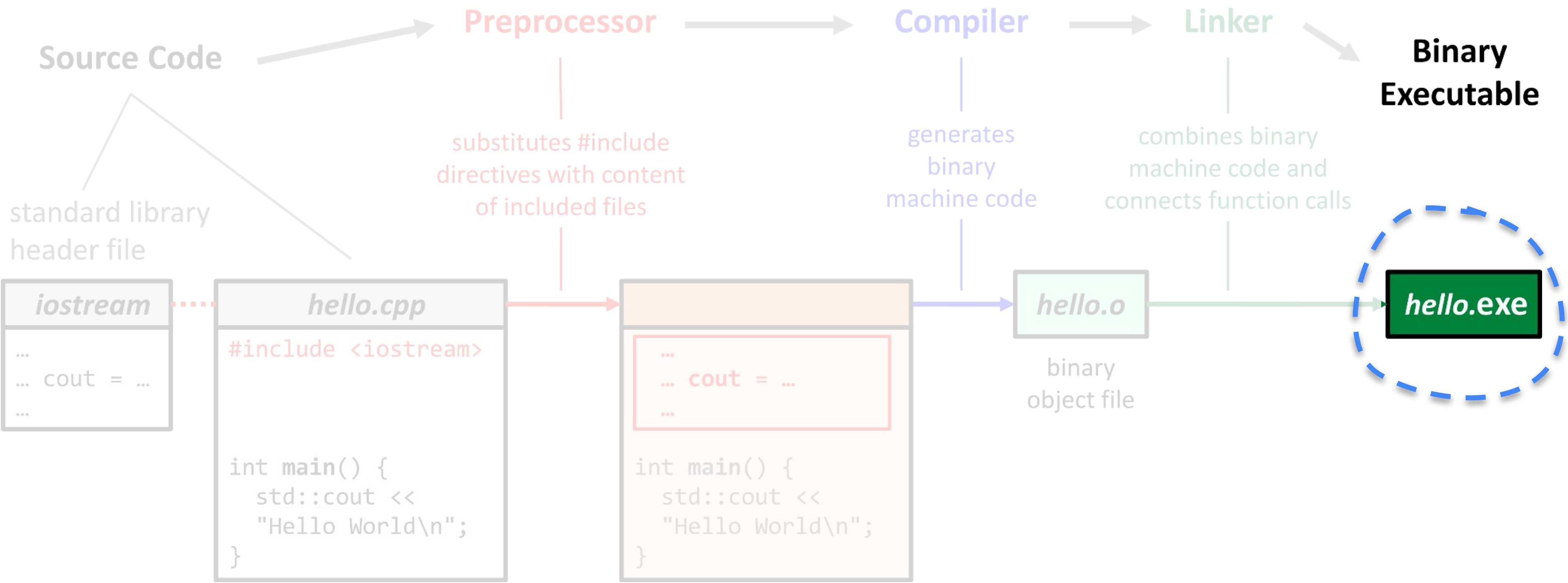
Last time on CS 4440…

Binary Reverse Engineering
Instruction Recovery
Control Flow Analysis
Structure Recovery
RE Challenges

Recap: the Compilation Process



Recap: the Compilation Process



Closed-source Software

- It's everywhere!

Closed-source Software

- It's everywhere!

Commercialized applications and libraries

Freely-distributed **proprietary software**

Legacy software whose source code is lost

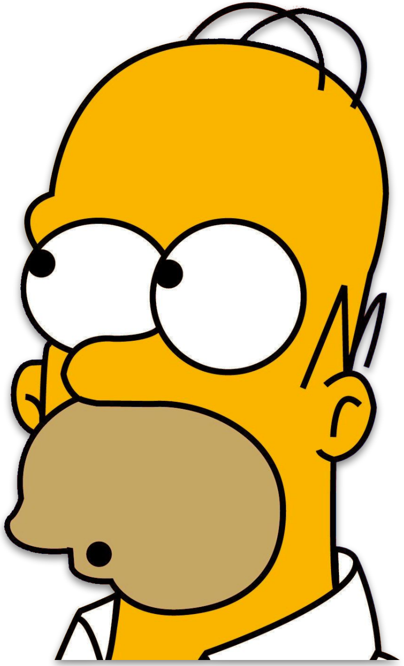
Reverse Engineering (RE)

- **What is RE?**

  "A process or method through which one attempts to **understand** through deductive reasoning how a previously made **device**, **process**, **system**, or piece of **software** accomplishes a task with **very little (if any) insight** into exactly how it does so."

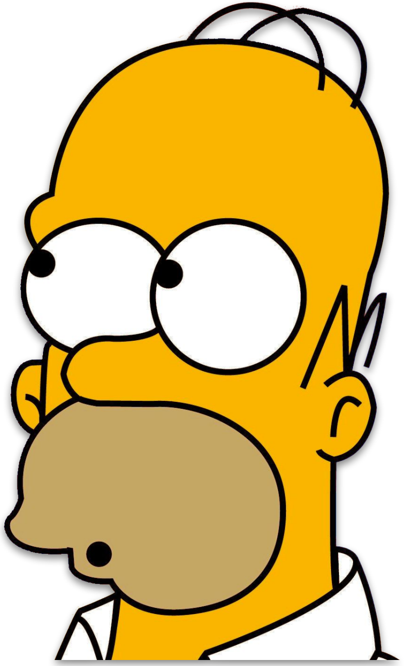
Three Pillars of RE

1. ???



Three Pillars of RE

1. **Instruction Recovery**



Pillar #1: Instruction Recovery

- **Goal: ???**

Pillar #1: Instruction Recovery

- **Goal:** translate bytes into **logical instructions**
  - Called instruction **decoding**
  - Analogous to what CPU does
  - General output: **disassembly**

Instruction stream

```
B8 22 11 00 FF 01 CA 31 F6 53 8B 5C 24
04 8D 34 48 39 C3 72 EB C3
```

Read bytes from input executable

Machine code bytes

```
B8 22 11 00 FF
01 CA
31 F6
53
8B 5C 24 04
8D 34 48
39 C3
72 EB
C3
```

Group bytes

Assembly language statements

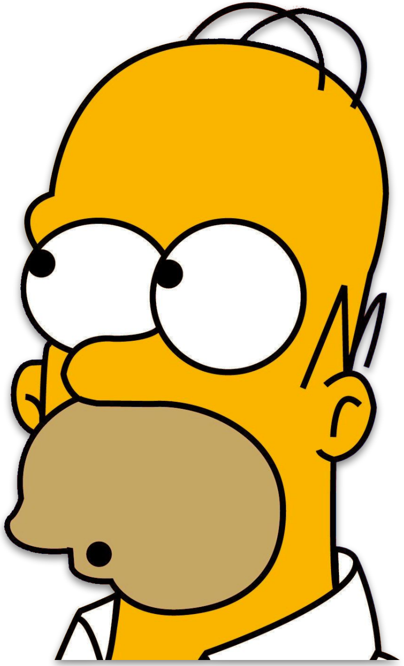
```
foo:
movl $0xFF001122, %eax
addl %ecx, %edx
xorl %esi, %esi
pushl %ebx
movl 4(%esp), %ebx
leal (%eax,%ecx,2), %esi
cmpl %eax, %ebx
jnae foo
retl
```

Decode instructions

Three Pillars of RE

1. **Instruction Recovery**
   - Decode bytes to instructions
   - Disambiguate code from data

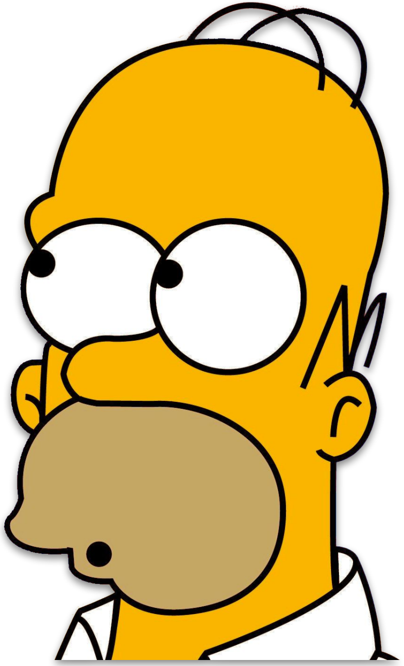
2. **???**



Three Pillars of RE

1. **Instruction Recovery**
   - Decode bytes to instructions
   - Disambiguate code from data

2. **Control Flow Recovery**
   - Intra-procedural execution flow
   - Inter-procedural execution flow



Pillar #2: Control Flow Recovery

- **Direct Edges**
  - **???**

Pillar #2: Control Flow Recovery

- **Direct Edges**
  - Jump/call a function
- **Indirect Edges**
  - **???**

```
jmp 0x4001AB3
```

Target is pre-set **statically**

Pillar #2: Control Flow Recovery

- **Direct Edges**
  - Jump/call a function
- **Indirect Edges**
  - Transfer to a register
  - Function pointers
  - Switch-case tables
- **"Pseudo" Edges**
  - **???**

`jmp 0x4001AB3`

Target is pre-set **statically**

`call %eax;` where?

Target found at **runtime**

Pillar #2: Control Flow Recovery

- **Direct Edges**
  - Jump/call a function
- **Indirect Edges**
  - Transfer to a register
  - Function pointers
  - Switch-case tables
- **"Pseudo" Edges**
  - Post-call returns
- **Tail Calls**
  - ???

`jmp 0x4001AB3`

Target is pre-set **statically**

`call %eax;` where?

Target found at **runtime**

`ret;` goes where?

Necessary to recover **all paths**

Pillar #2: Control Flow Recovery

- **Direct Edges**
  - Jump/call a function
- **Indirect Edges**
  - Transfer to a register
  - Function pointers
  - Switch-case tables
- **"Pseudo" Edges**
  - Post-call returns
- **Tail Calls**
  - Call at function's end

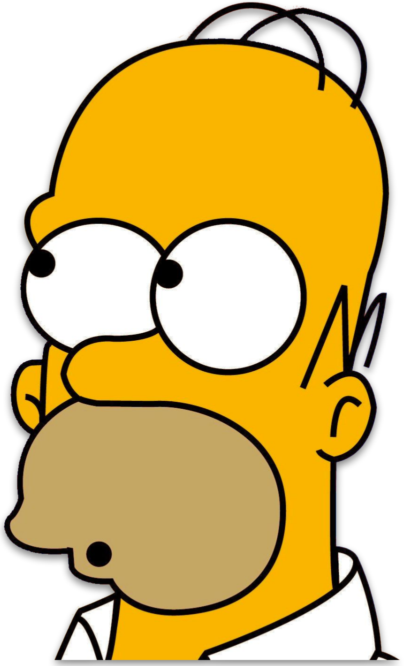
`jmp 0x4001AB3`	Target is pre-set **statically**
`call %eax;` where?	Target found at **runtime**
`ret;` goes where?	Necessary to recover **all paths**
`jmp &foo;` call?	Expressed as **jumps**, not calls

Three Pillars of RE

1. **Instruction Recovery**
   - Decode bytes to instructions
   - Disambiguate code from data

2. **Control Flow Recovery**
   - Intra-procedural execution flow
   - Inter-procedural execution flow

3. **???**

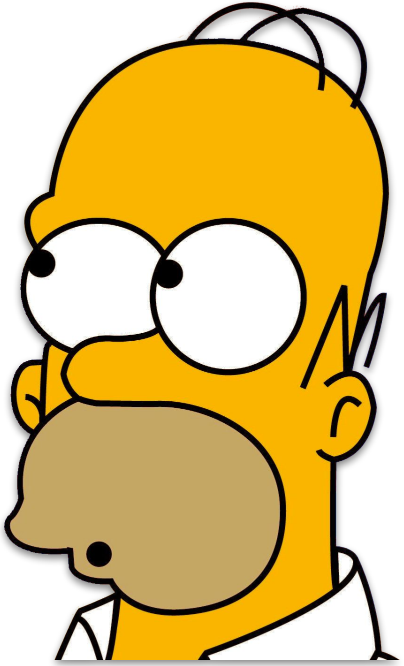


Three Pillars of RE

1. **Instruction Recovery**
   - Decode bytes to instructions
   - Disambiguate code from data

2. **Control Flow Recovery**
   - Intra-procedural execution flow
   - Inter-procedural execution flow

3. **Program Structure Recovery**
   - Identify program basic blocks
   - Higher-level constructs (e.g., loops)



Pillar #3: Structure Recovery

- Largely **heuristic**-based
  - Construct-specific rules
- **Functions:**
  - **Start:**
    - **???**

Pillar #3: Structure Recovery

- Largely **heuristic**-based
  - Construct-specific rules
- **Functions:**
  - **Start:**
    - Target of a `call`
    - Target of a tail call
    - A known prologue
    - A dispatch table entry
  - **End:**
    - **???**

```
push ebp
mov ebp, esp
sub esp, N
```

Prologue

```
switch(choice) {
    case 0 :
        result = add(first, second);
        break;
    case 1 :
        result = sub(first, second);
        break;
    case 2 :
        result = mult(first, second);
        break;
    case 3 :
        result = divide(first, second);
        break;
}
```

C-level Switch Table

Pillar #3: Structure Recovery

- Largely **heuristic**-based
  - Construct-specific rules

- **Functions:**
  - **Start:**
    - Target of a `call`
    - Target of a tail call
    - A known prologue
    - A dispatch table entry
  - **End:**
    - Location of a `ret`
    - Location of a tail call
    - A known epilogue

```
push ebp
mov ebp, esp
sub esp, N
```

Prologue

```
mov esp, ebp
pop ebp
ret
```

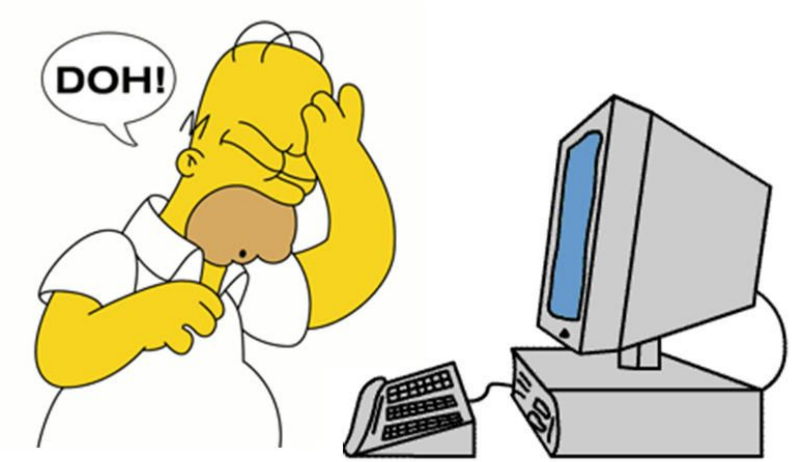
Epilogue

```
switch(choice) {
    case 0 :
        result = add(first, second);
        break;
    case 1 :
        result = sub(first, second);
        break;
    case 2 :
        result = mult(first, second);
        break;
    case 3 :
        result = divide(first, second);
        break;
}
```

C-level Switch Table

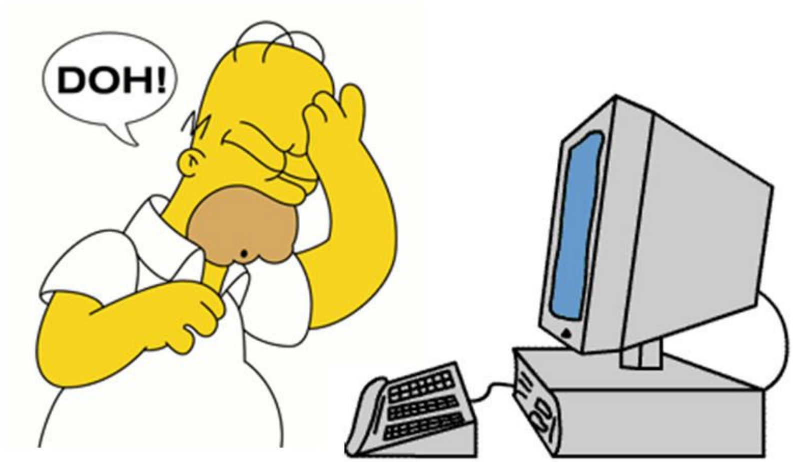
Challenges to RE

- ???



Challenges to RE

- **Compiler Craziness**
  - Data-in-code
  - Optimizations
- **Haphazard Heuristics**
  - Weird/esoteric patterns
  - E.g., all jump table variants
- **Obtuse Obfuscations**
  - Control-flow flattening
  - Opaque predicates



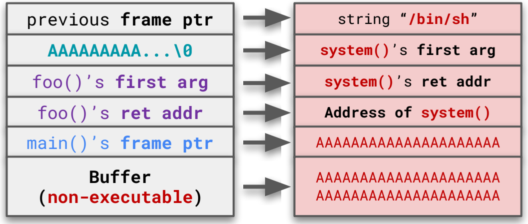
Questions?

This time on CS 4440...

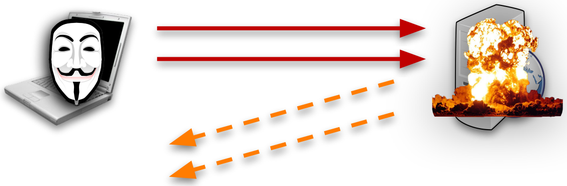
Side Channels
Hardware Security
Hardware Supply Chain Attacks

Exploitable Security Flaws

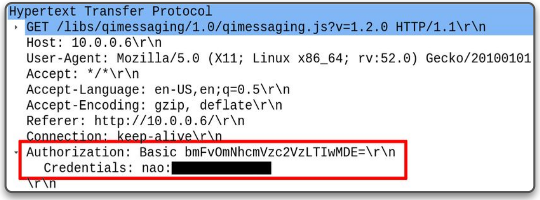
- So far, we have studied attacks that exploit **design** flaws



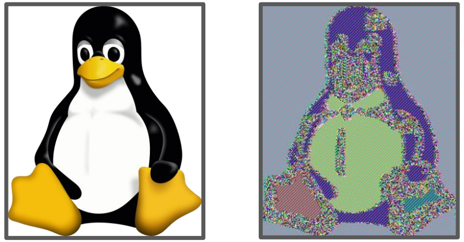
Buffer Overflows



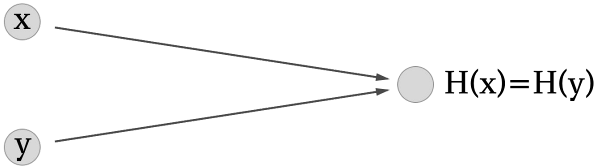
SYN Flooding



Sniffing Unencrypted Data



ECB Diffusion Analysis



Hash Collisions

```
http://cs4440.eng.utah
.edu/project3/search
?q=%3Cscript%3E...
```

Cross-site Scripting

Exploitable Security Flaws

- So far, we have studied attacks that exploit design flaws

ECB Diffusion Analysis

Hash Collisions

Cross-site Scripting

What if I told you that **implementation flaws** can be just as severe?

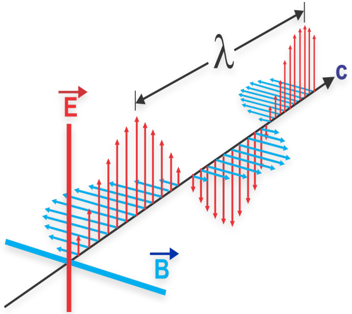
Side Channel Attacks

Side Channel Attacks

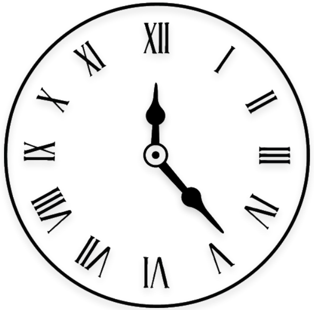
"Any attack based on **extra information** that can be **gathered** because of the fundamental way a computer protocol or algorithm is **implemented**, or minor, but potentially devastating, mistakes or oversights in the implementation."

Side Channels

- What are some potential sources of **indirect info** emitted by your computer?
  - **Additional channels** of information beyond what is directly visible/accessible to you



Emitted Radiation

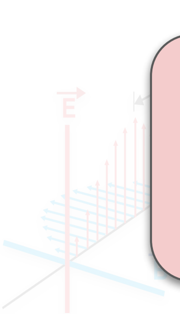


Execution Time

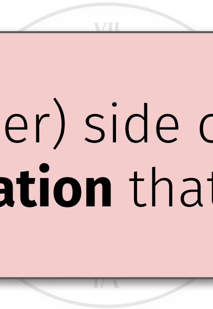
Power Consumption

Side Channels

- What are some potential sources of **indirect info** emitted by your computer?
  - **Additional channels** of information beyond what is directly visible/accessible to you



Emitted Radiation



Execution Time

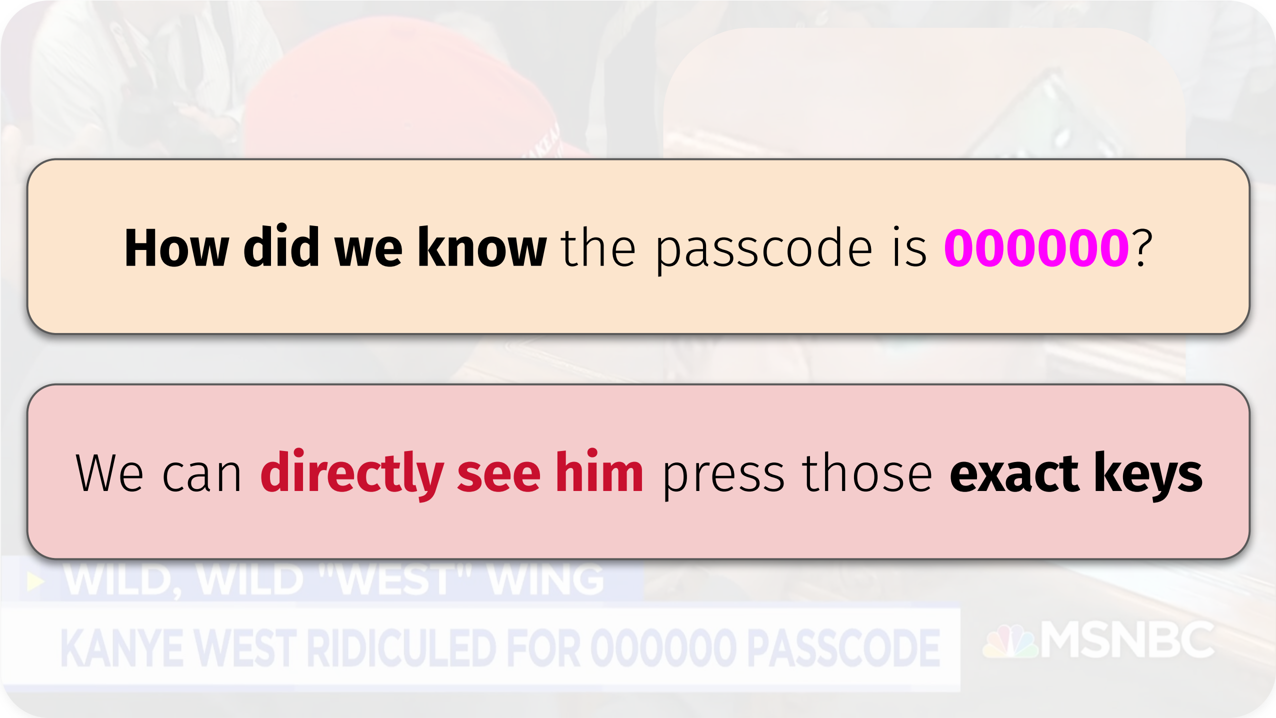
Power Consumption

These (and other) side channels reveal **critical information** that is **exploitable**

Optical and Acoustic Side Channels

Stealing Passwords

Stealing Passwords



How did we know the passcode is **000000**?

We can **directly see him** press those **exact keys**

Stealing Passwords

- What if we **can't directly see** keys that someone is pressing?

Stealing Passwords

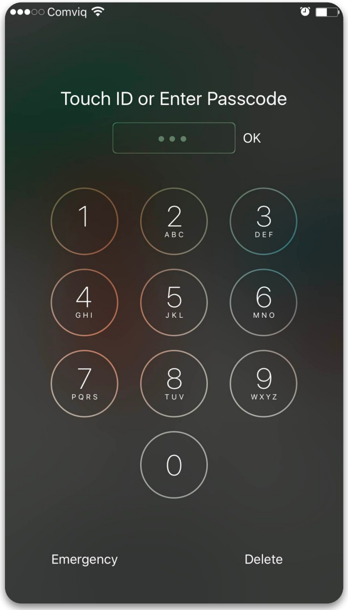
- What if we **can't directly see** keys that someone is pressing?

- **Optical side channel:**
  - Capture visible **hand movements**

Stealing Passwords

- What if we **can't directly see** keys that someone is pressing?

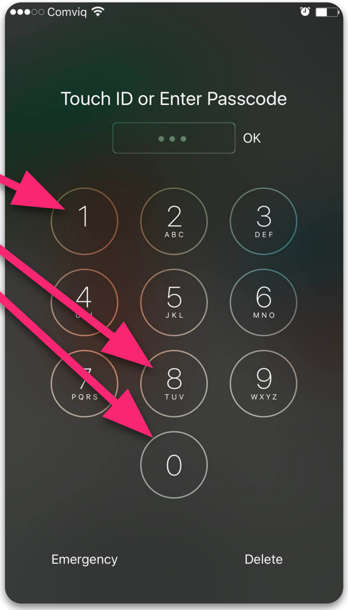
- **Optical side channel:**
  - Capture visible **hand movements**
  - Assume attacker **knows (or can easily guess)** the key interface



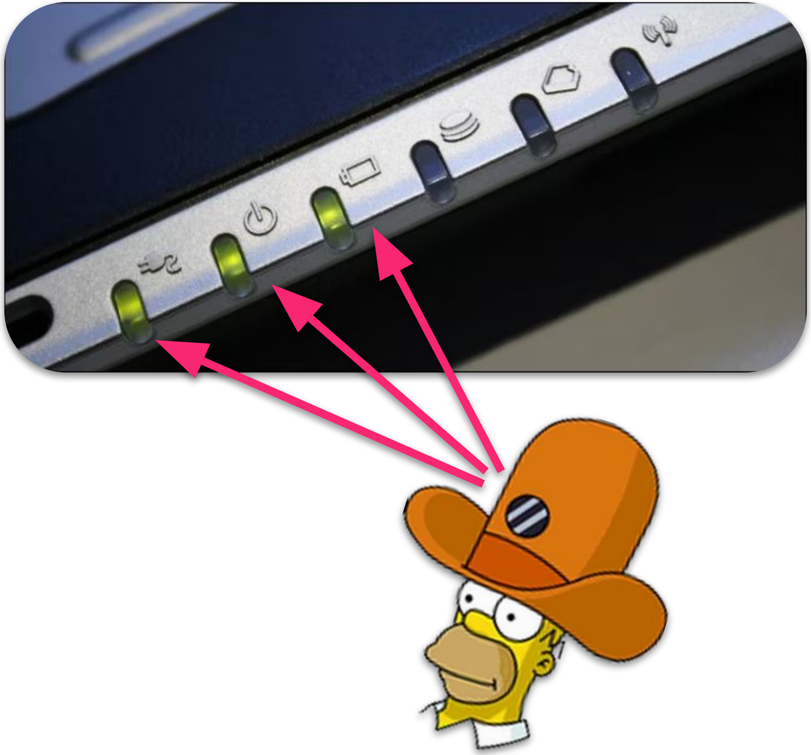
Stealing Passwords

- What if we **can't directly see** keys that someone is pressing?

- **Optical side channel:**
  - Capture visible **hand movements**
  - Assume attacker **knows (or can easily guess)** the key interface
  - Attacker **maps movements** to pressed keys on the interface



Stealing Information



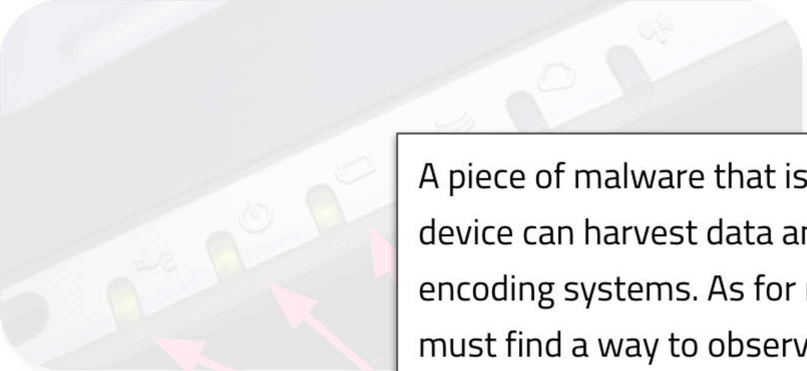
Hard Drive LED Allows Data Theft From Air-Gapped PCs

Researchers at Ben-Gurion University of the Negev in Israel have disclosed yet another method that can be used to exfiltrate data from air-gapped computers, and this time it involves the activity LED of hard disk drives (HDDs).

Researchers at Ben-Gurion University of the Negev in Israel have disclosed yet another method that can be used to exfiltrate data from air-gapped computers, and this time it involves the activity LED of hard disk drives (HDDs).

Many desktop and laptop computers have an HDD activity indicator, which blinks when data is being read from or written to the disk. The blinking frequency and duration depend on the type and intensity of the operation being performed.

Stealing Information



Hard Drive LED Allows Data Theft From Air-Gapped PCs

A piece of malware that is installed on the targeted air-gapped device can harvest data and exfiltrate it using one of these encoding systems. As for reception and decoding, the attacker must find a way to observe the targeted device's activity LED, either using a local hidden camera, a high-resolution camera that can capture images from outside the building, a camera mounted on a drone, a compromised security camera, a camera carried by a malicious insider, or optical sensors.

Acoustic Side Channels

- **Sound** can leak information, too!
  - Keyboard enthusiasts beware

Acoustic Side Channels

- **Sound** can leak information, too!
  - Keyboard enthusiasts beware
- **Build model of key press noises**
  - Model refinement:
    - **???**

Acoustic Side Channels

- **Sound** can leak information, too!
  - Keyboard enthusiasts beware

- **Build model of key press noises**
  - Model refinement:
    - Consider microphone
    - Remove ambient noise
  - Use model to infer entered data
    - Passwords
    - Usernames
    - Phone numbers

Questions?

Timing Side Channels

Password Checking

- **Password verification**—how would you implement this?

```
bool checkPW(char *testPW, char *realPW, int len) {
    for (int i = 0; i < len; i++) {
        if (testPW[i] != realPW[i]) {
            return false;
        }
    }
    return true;
}
```

Analogous to memcmp( )

Password Checking

- **Password verification**—how would you implement this?

```
bool checkPW(char *testPW, char *realPW, int len) {

    for (int i = 0; i < len; i++) {

        if (testPW[i] != realPW[i]) {
            return false;
        }
    }

    return true;
}
```

Analogous to memcmp( )

Does this password checking code reveal a **security flaw**?

Does this password-checking code reveal a security flaw?

No—an attacker could only brute-force guess!

0%

Yes—the design is vulnerable (e.g., buffer overflow).

0%

None of the above

0%

```
bool checkPW(char *testPW, char *realPW, int len) {

    for (int i = 0; i < len; i++) {

        if (testPW[i] != realPW[i]) {
            return false;
        }
    }

    return true;
}
```

Password Checking

- **Password verification**—how would you implement this?

```
bool checkPW(char *testPW, char *realPW, int len) {

    for (int i = 0; i < len; i++) {

        if (testPW[i] != realPW[i]) {
            return false;
        }
    }

    return true;
}
```

Password Login Attempts:

ABCDEFGH == PASSWORD

- ???

Password Checking

- **Password verification**—how would you implement this?

```
bool checkPW(char *testPW, char *realPW, int len) {

    for (int i = 0; i < len; i++) {

        if (testPW[i] != realPW[i]) {
            return false;
        }
    }

    return true;
}
```

Password Login Attempts:

ABCDEFGH == PASSWORD
- **False** on first iteration

PASSEFGH == PASSWORD
- **???**

Password Checking

- **Password verification**—how would you implement this?

```
bool checkPW(char *testPW, char *realPW, int len) {

    for (int i = 0; i < len; i++) {

        if (testPW[i] != realPW[i]) {
            return false;
        }
    }

    return true;
}
```

Password Login Attempts:

ABCDEFGH == PASSWORD

- **False** on first iteration

PASSEFGH == PASSWORD

- **True** on iterations **1–4**
- **False** on **fifth** iteration

More code executed for a **correct** symbol!

Password Checking

How can this **side channel** be **exploited**?

Password Checking

How can this **side channel** be **exploited**?

Attacker: ABCDEF

Password Checking

How can this **side channel** be **exploited**?

Attacker: ABCDEF

Server: `False`
Server took **1ms** to respond

Password Checking

How can this **side channel** be **exploited**?

“C” took longer!

Attacker: ABCDEF

Server: False
Server took **1ms** to respond

Attacker: CBCDEF

Server: False
Server took **2ms** to respond

Password Checking

How can this **side channel** be **exploited**?

Attacker: CRCDEF

Server: False
Server took **2ms** to respond

Password Checking

How can this **side channel** be **exploited**?

Attacker: CRCDEF

Server: False
Server took **2ms** to respond

Attacker: CHIDEF

Server: False
Server took **4ms** to respond

Password Checking

How can this **side channel** be **exploited**?

Attacker: `CHIEFS`

Server: `True`
Server took **7ms** to respond

Password Checking

How can this **side channel** be **exploited**?

Attacker: `CHIEFS`

Server: `True`
Server took **7ms** to respond

Through **timing analysis**, attacker can infer the **correctness** of individual **password symbols**!

Password Checking

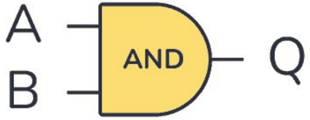
- **Solution:**
  - **???**

Password Checking

- **Solution:**
  - **Constant-time** implementation (e.g., using bitwise **AND**-ing)

```
bool checkPW(char *testPW, char *realPW, int len) {
    bool result = 1; // integer equiv of "true"
    for (int i = 0; i < len; i++) {
        result &= ca[i] == cb[i];
        return result;
    }
}
```

Guess: PASSEFGH
Bit: 11110000
Result: **False**



A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

Password Checking

- **Solution:**
  - **Constant-time** implementation (e.g., using bitwise **AND**-ing)

```
bool checkPW(char *testPW, char *realPW, int len) {
    bool result = 1; // integer equiv of "true"
    for (int i = 0; i < len; i++) {
        result &= ca[i] == cb[i];
        return result;
    }
}
```

Guess: PASSEFGH
Bit: 11110000
Result: **False**

Password Login Attempts:

ABCDEFGH == PASSWORD
- **False** on **last** iteration

PASSEFGH == PASSWORD
- **False** on **last** iteration

PASSWORD == PASSWORD
- **True** on **last** iteration

True and **False** run for **identical time**!

Password Checking

- **Implications:**
  - **???**

Password Checking

- **Implications:**
  - **Never** use **timing-unsafe** **string compares** when handling **sensitive data**!

FreeBSD Manual Pages

timingsafe_bcmp | man | apropos

3 - Subroutines | FreeBSD 13.1-RELEASE and Ports | All Architectures | html

home | help

```
TIMINGSAFE_BCMP(3)      FreeBSD Library Functions Manual      TIMINGSAFE_BCMP(3)

NAME
     timingsafe_bcmp, timingsafe_memcmp -- timing-safe byte sequence compar-
     isons

SYNOPSIS
     #include <string.h>

     int
     timingsafe_bcmp(const void *b1, const void *b2, size_t len);

     int
     timingsafe_memcmp(const void *b1, const void *b2, size_t len);
```

FreeBSD Manual Pages

consttime_memequal | man | apropos

All Sections | NetBSD 7.0 | All Architectures | html

home | help

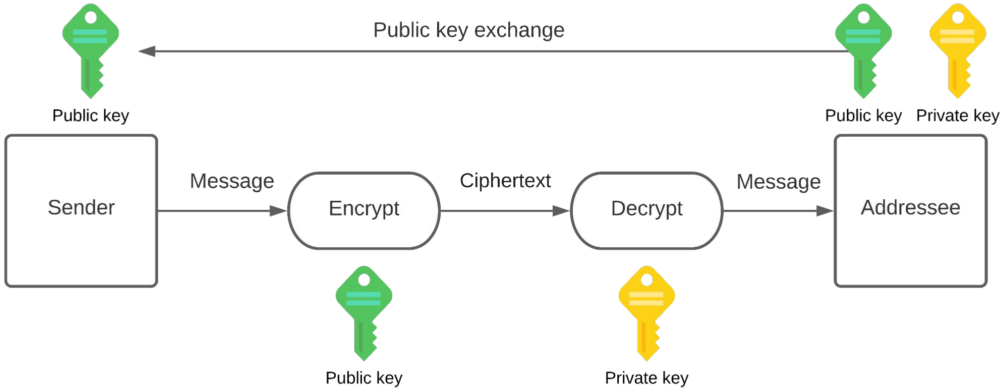
```
CONSTTIME_MEMEQUAL(3)    BSD Library Functions Manual    CONSTTIME_MEMEQUAL(3)

NAME
     consttime_memequal -- compare byte strings for equality without timing
     leaks

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <string.h>

     int
     consttime_memequal(void *b1, void *b2, size_t len);
```

Questions?

Power Side Channels

Recap: RSA Encryption

- **Summary:**
  - ???

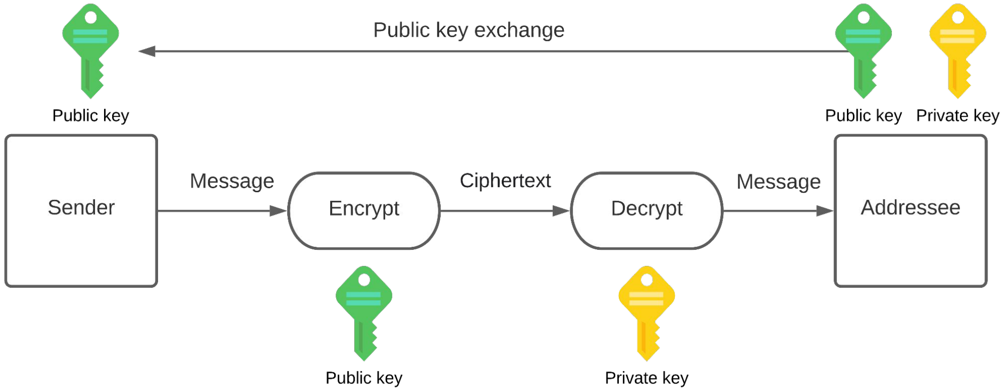


Recap: RSA Encryption

- **Summary:**
  - Encrypt with public key
  - Decrypt with private key
  - Public key = (**e**,**N**)
  - Private key = (**d**,**N**)

- To **encrypt:**
  - $E(x) = x^{e} \bmod N$

- To **decrypt:**
  - $D(x) = x^{d} \bmod N$



Recap: RSA Encryption

- **Summary:**
  - Encrypt with public key
  - Decrypt with private key
  - Public key = (e,N)
  - Private key = (d,N)

- **To encrypt:**
  - $E(x) = x^e \bmod N$
- **To decrypt:**
  - $D(x) = x^d \bmod N$

Modular exponentiation must be implemented **efficiently**

Modular Exponentiation

- **Decryption:** $D(x) = C^{privKey} \bmod N$

```
x = C

for (int i = 0; i < len; i++){

    x = (x·x) mod(N)

    if (privKey[i] == 1){
        x = (x·C) mod(N)
    }
}
return x
```

Does this decryption code reveal a **security flaw**?

Does this decryption code reveal a security flaw?

No—still would have to brute-force the PrivKey!

0%

Yes—more/fewer operations on different key bits!

0%

None of the above

0%

```
x = C

for (int i = 0; i < len; i++){

     x = (x·x) mod(N)

     if (privKey[i] == 1){
           x = (x·C) mod(N)
     }
}
return x
```

Modular Exponentiation

- **Decryption:** $D(x) = C^{privKey} \bmod N$

```
x = C

for (int i = 0; i < len; i++){

    x = (x·x) mod(N)

    if (privKey[i] == 1){
        x = (x·C) mod(N)
    }
}
return x
```

Bit-specific Operations:

`privKey[i] == 0`

1. Find square of **x**
2. Take modulo **N**

`privKey[i] == 1`

1. Find square of **x**
2. Take modulo **N**

Modular Exponentiation

- **Decryption:** $D(x) = C^{privKey} \bmod N$

```
x = C

for (int i = 0; i < len; i++){

    x = (x·x) mod(N)

    if (privKey[i] == 1){
        x = (x·C) mod(N)
    }
}
return x
```

Bit-specific Operations:

`privKey[i] == 0`

1. Find square of **x**
2. Take modulo **N**

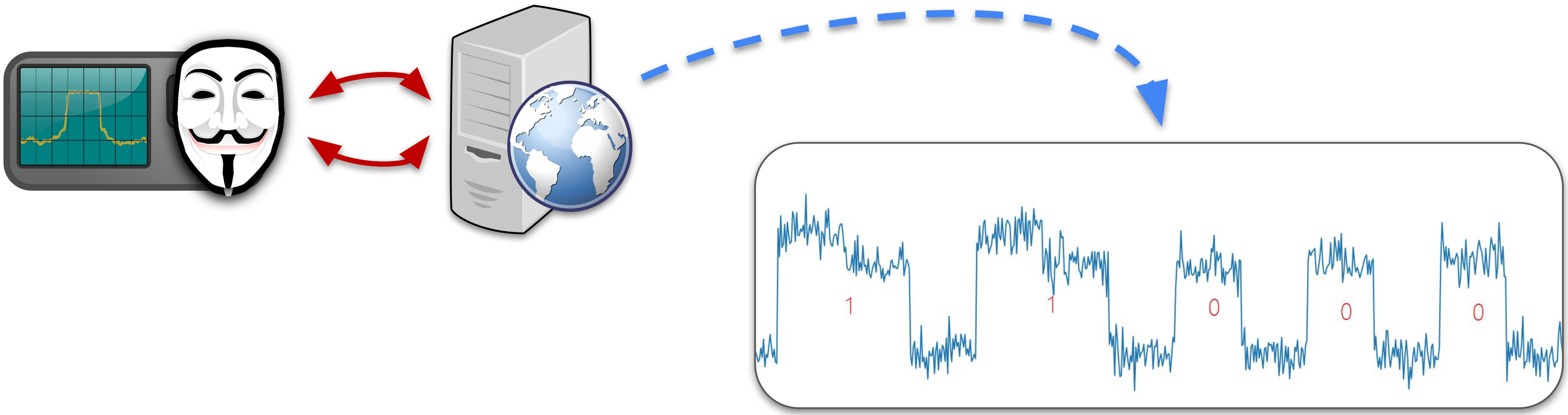
`privKey[i] == 1`

1. Find square of **x**
2. Take modulo **N**
3. Multiply by **C**
4. Take modulo **N**

Timing and **power** will **differ** between key bits **0** versus **1**!

RSA Power Analysis

How can this **side channel** be **exploited**?

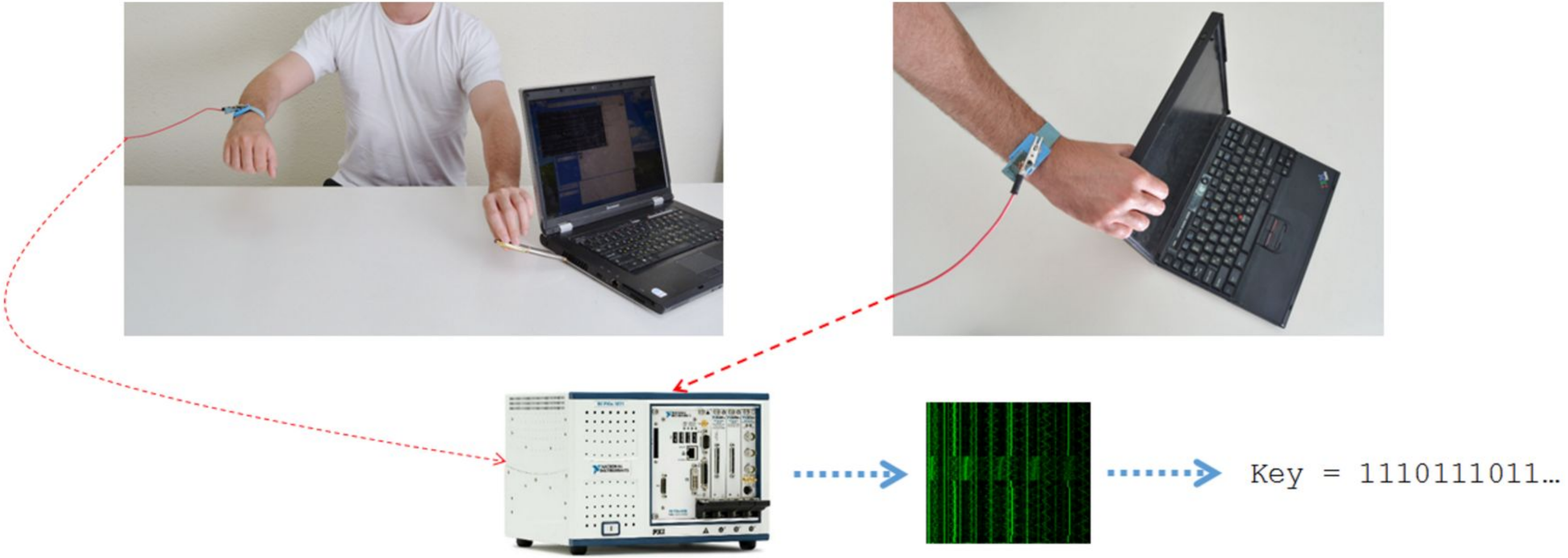


RSA Power Analysis

How can this **side channel** be **exploited**?

Attacker can retrieve a user's private key!

Realistic Power Analysis



Questions?

Cache-based Side Channels

CPU Caches

- **RAM** is expensive to load from
  - **Disk** is even more expensive!
- Fastest retrieval: **???**

Storage	Read Time	Capacity	Managed By
Hard Disk	10ms	1 TB	Software/OS
Flash Drive	10–100us	100 GB	Software/OS
RAM	200 cycles	10 GB	Software/OS

https://computationstructures.org/lectures/caches/caches.html

CPU Caches

- **RAM** is expensive to load from
  - **Disk** is even more expensive!
- Fastest retrieval: the **CPU cache**
  - Small storage built-in to CPU
  - Common hierarchy: L1, L2, L3, L4
- Key purpose: accelerate retrieval of **commonly-accessed data**

Storage	Read Time	Capacity	Managed By
Hard Disk	10ms	1 TB	Software/OS
Flash Drive	10–100us	100 GB	Software/OS
RAM	200 cycles	10 GB	Software/OS
L3 Cache	40 cycles	10 MB	Hardware
L2 Cache	10 cycles	256 KB	Hardware
L1 Cache	2–4 cycles	32 KB	Hardware

https://computationstructures.org/lectures/caches/caches.html

Program Execution

- What do you expect to happen here?
  - `index < arraySize`
    - ???

```
int read(int index){
    int result = -1;
    result = array[index];
    return result;
}
```

Program Execution

- What do you expect to happen here?
  - `index < len(array)`
    - Within-bounds read... **success**
  - `index > len(array)`
    - **???**

```
int read(int index){
    int result = -1;
    result = array[index];
    return result;
}
```

Program Execution

- What do you expect to happen here?
  - `index < len(array)`
    - Within-bounds read... **success**
  - `index > len(array)`
    - Out-of-bounds read... **prevent**
- Optimization: **Speculative Execution**
  - Perform the **OOB read** anyways

```
int read(int index){
    int result = -1;
    result = array[index];
    return result;
}
```

Program Execution

- What do you expect to happen here?
  - `index < len(array)`
    - Within-bounds read... **success**
  - `index > len(array)`
    - Out-of-bounds read... **prevent**

- Optimization: **Speculative Execution**
  - Perform the **OOB read** anyways
  - **Cache** whatever data is accessed
  - Check if it's allowed... **after the fact**
  - **Roll-back** the cache to correct state

```
int read(int index){
    int result = -1;
    result = array[index];
    return result;
}
```

Save time by having data **pre-cached** and ready to go!

Program Execution

- What do you expect to happen here?
  - index <
    - Wi
  - index
    - Ou
- Optimizati
  - Perform
  - Cache w
  - Check if
  - Roll-back the cache to correct state

```
int read(int index){
        1;
        y[index];
        ;
}
```

aving data
ready to go!

Implication: data we **shouldn't** have access to (e.g., from another program) is **cached**

Cache lookup is faster… can we exploit a **timing side channel** to recover this data?

Attacking Speculative Execution

- Suppose speculative execution caches a secret `result` of **`4440`**

```
// index > len(array)
int read(int index){
    int result = -1;
    result = array[index];
    return result;
}
```

Attacking Speculative Execution

- Suppose speculative execution caches a secret `result` of **`4440`**

```
// index > len(array)
int read(int index){
     int result = -1;
     result = array[index];
     return result;
}
```

1. Cache `array[index]`
2. Bounds check `index`
3. **Clear `array[index]`**

Due to roll-back, we can't retrieve result!

Attacking Speculative Execution

- Suppose speculative execution caches a secret `result` of `4440`

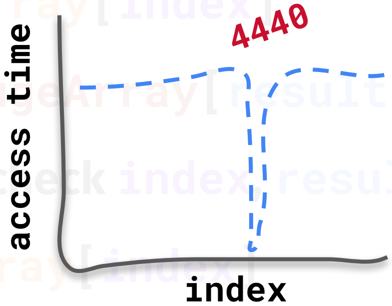
```
// index > len(array)
int read(int index){
    int result = -1;
    result = array[index];
    int dummy = hugeArray[result];
    return result;
}
```

1. Cache `array[index]`
2. Cache `hugeArray[result]`
3. Bounds check `index`, `result`
4. Clear `array[index]`
5. `hugeArray[result]` stays...

Attacking Speculative Execution

How can attacker figure out `result` is `4440`?

```
for (int i=0; i<...; i++){
    int x = hugeArray[i];
}
```



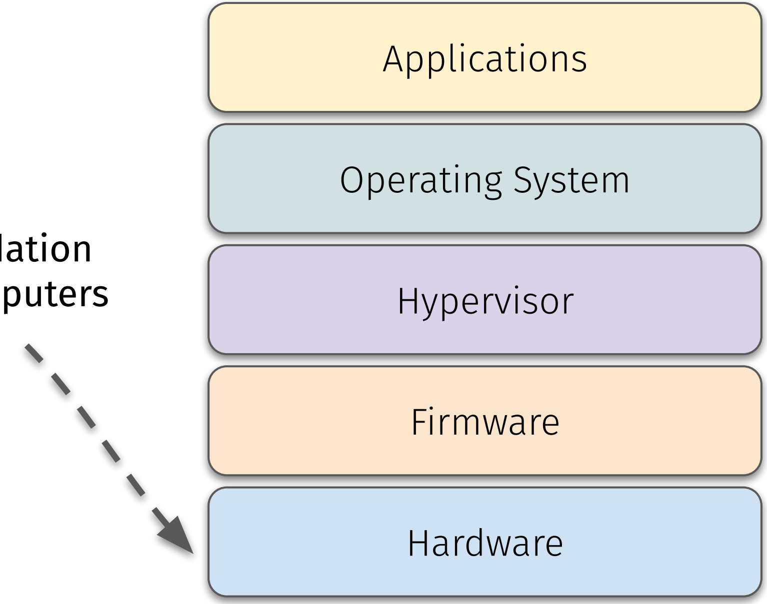
Since 4440 was cached, `hugeArray[4440]` has the fastest access time of all array indices!

Questions?

Hardware Security

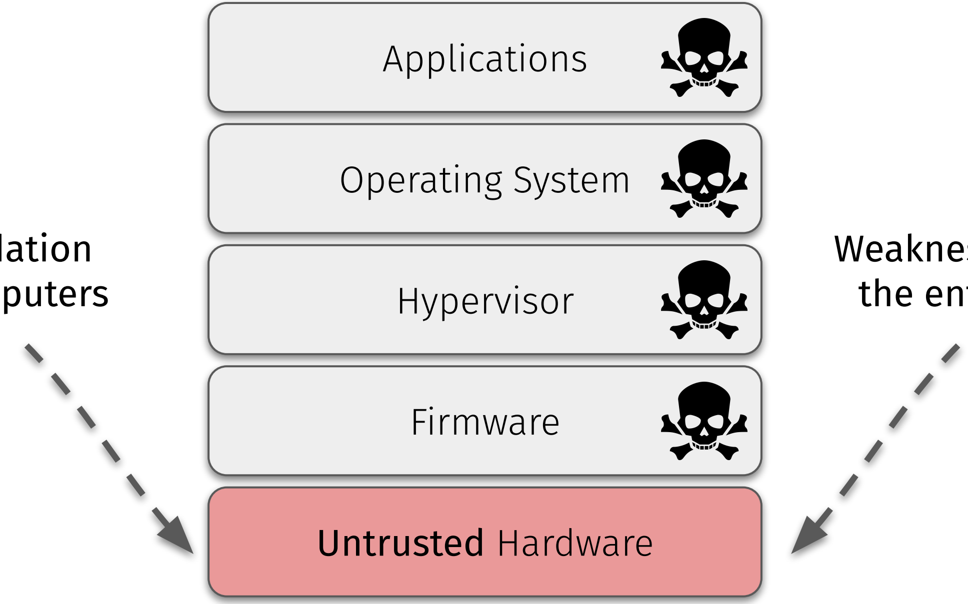
Hardware

The foundation of our computers



Hardware

The foundation
of our computers



Weaknesses weaken
the entire system

Hardware

FORESHADOW

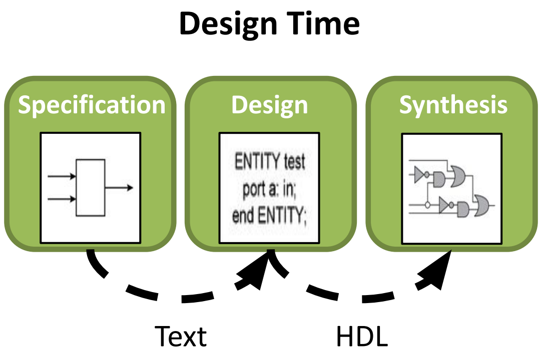
The foundation of our computers

Weaknesses weaken the entire system

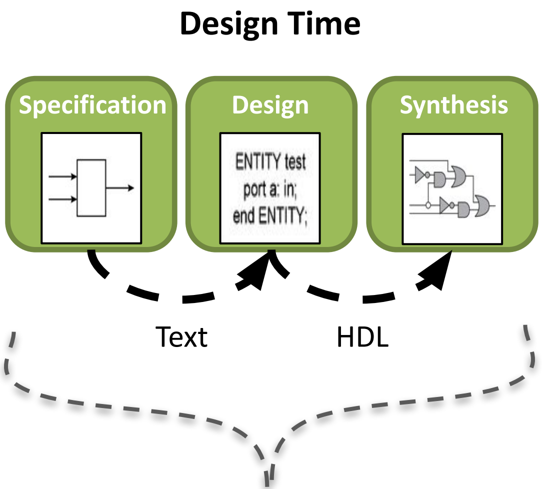
MELTDOWN

SPECTRE

Creating Hardware

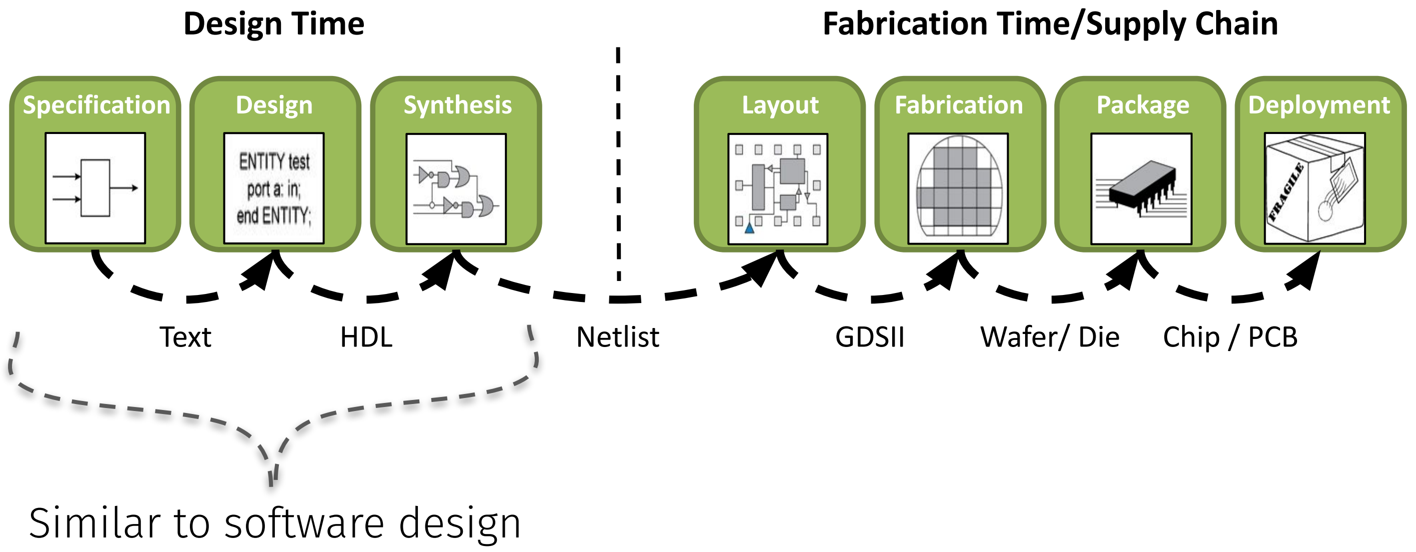


Creating Hardware

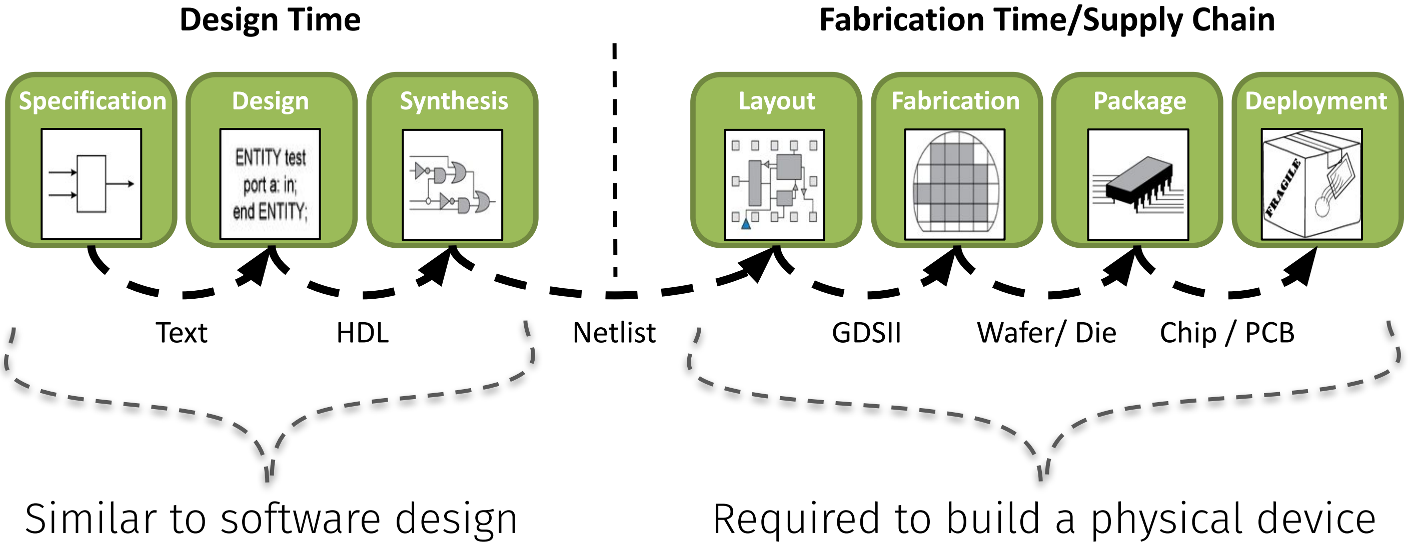


Similar to software design

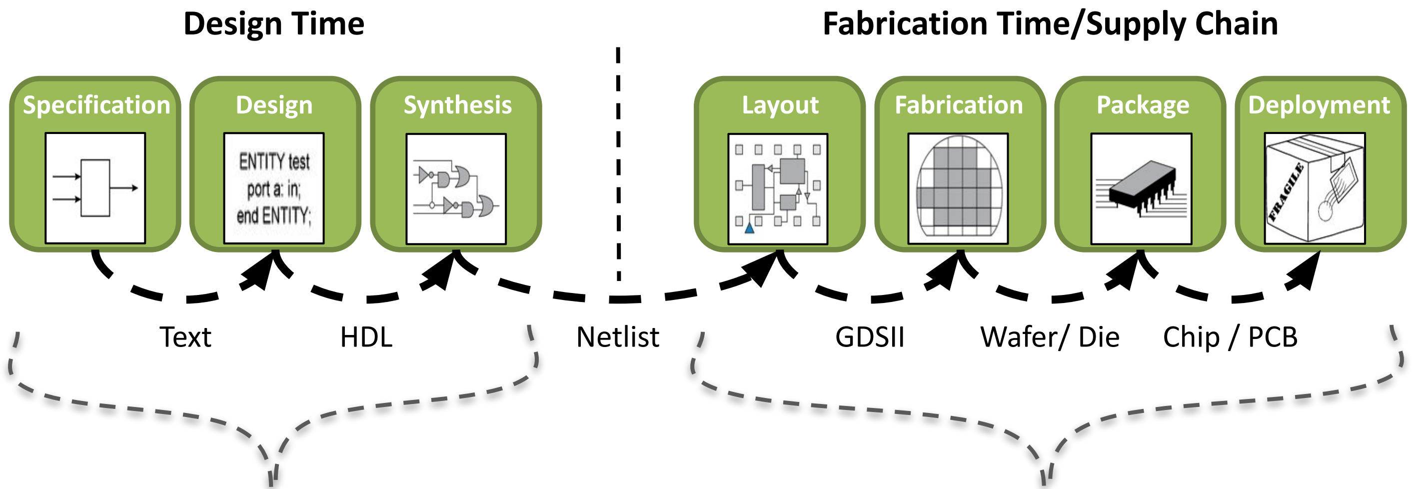
Creating Hardware



Creating Hardware



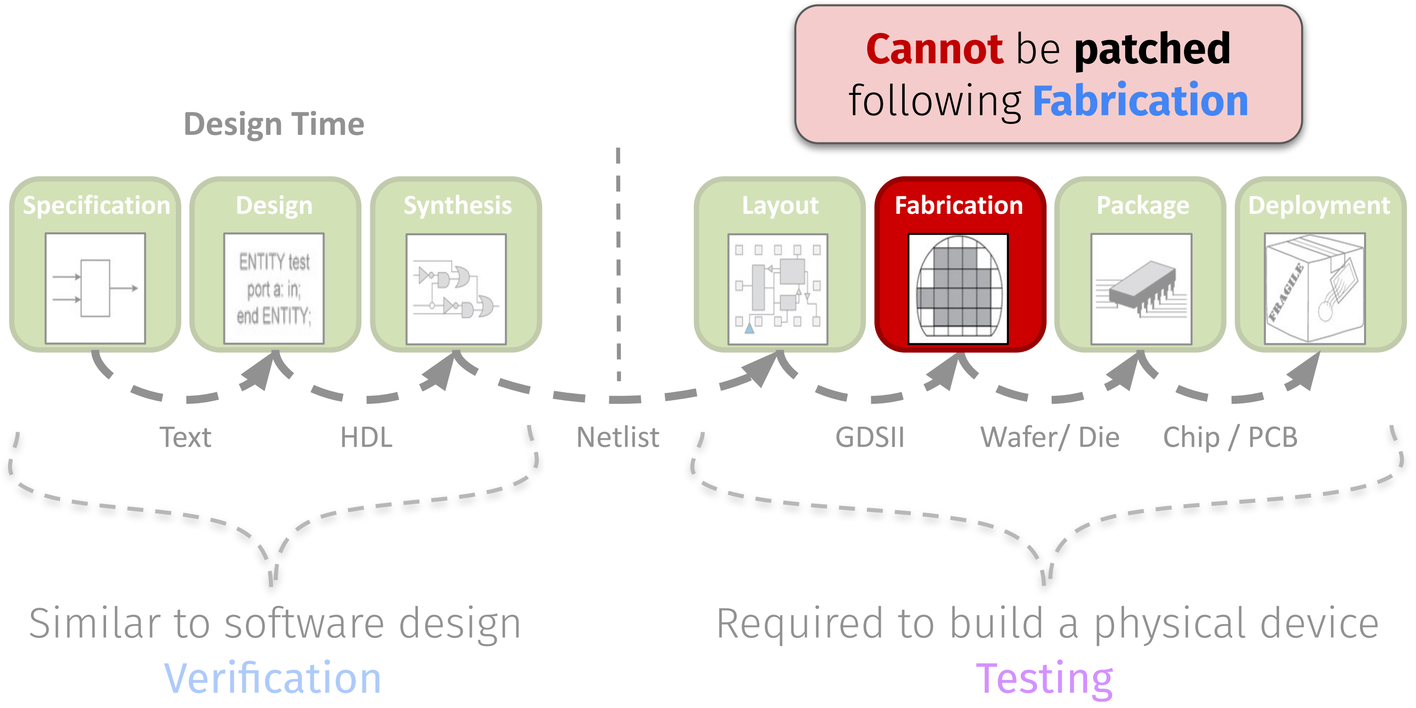
Creating Hardware



Similar to software design
Verification

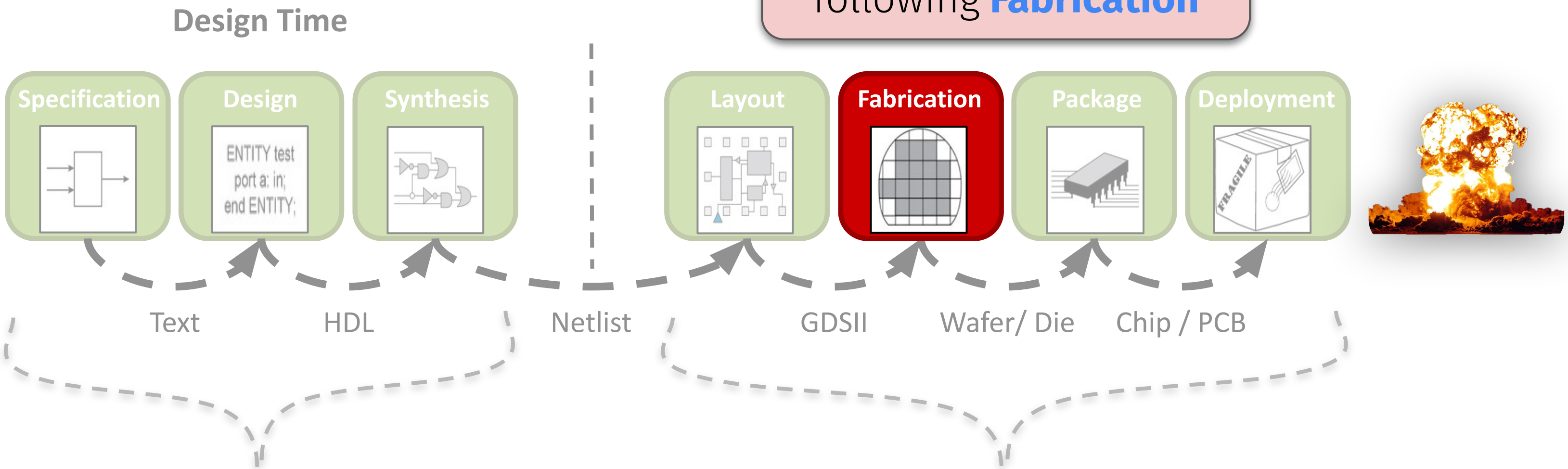
Required to build a physical device
Testing

Hardware Bugs



Hardware Bugs

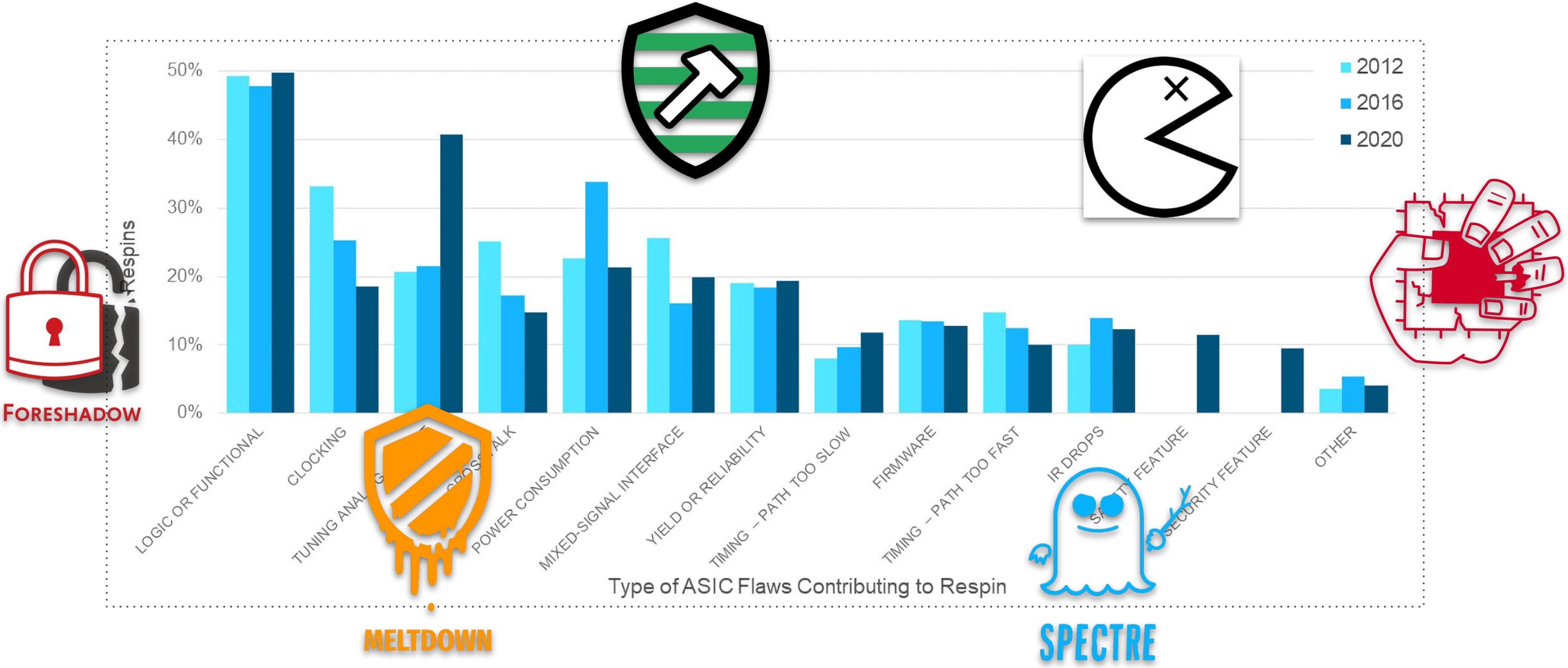
Cannot be **patched**
following **Fabrication**



Similar to software design
Verification

Required to build a physical device
Testing

Hardware Bugs



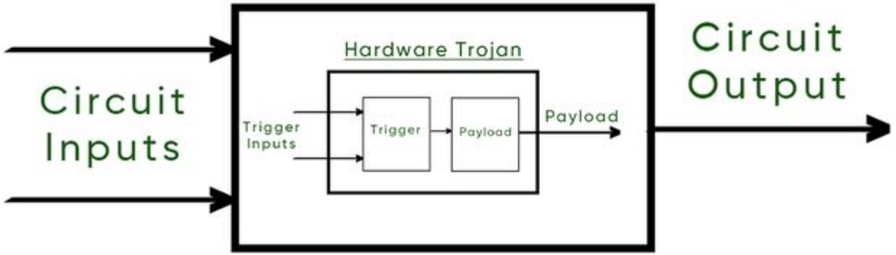
Hardware Threats

Hardware Trojans

- **Trojan Horse:**
  - **???**

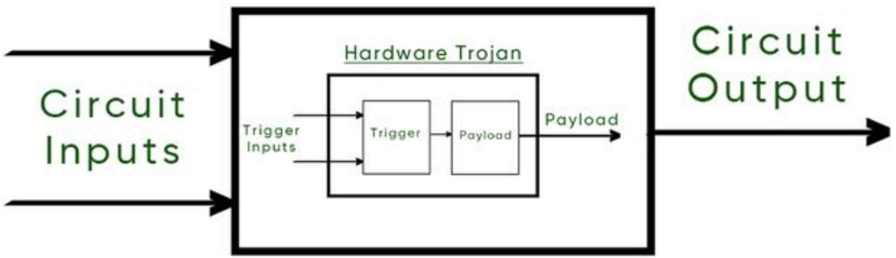
Hardware Trojans

- **Trojan Horse:**
  - Attack pre-inserted into chip
  - Will be **exploited** at **run time**
  - **Remotely triggered** by attacker



Hardware Trojans

- **Trojan Horse:**
  - Attack pre-inserted into chip
  - Will be **exploited** at **run time**
  - **Remotely triggered** by attacker



- **Ideal characteristics:**
  - Small
  - Stealthy
  - Controllable

Hardware Trojans

- **Trojan Horse:**
  - Attack pre-inserted into chip
  - Will be **exploited** at **run time**
  - **Remotely triggered** by attacker
- **Ideal characteristics:**
  - Small
  - Stealthy
  - Controllable
- **Engineering a trigger**

```
void attack_signed_c() {
    volatile int a, b, c = 0;

    while(1) {
        int c1 = c;
        int b1 = b;

        int i1 = ((b1 / c1) + 1);
        int i2 = ((i1 / c1) + 1);
        int i3 = ((i2 / c1) + 1);
        int i4 = ((i3 / c1) + 1);
        int i5 = ((i4 / c1) + 1);
        int i6 = ((i5 / c1) + 1);
        int i7 = ((i6 / c1) + 1);
        int i8 = ((i7 / c1) + 1);
        int i9 = ((i8 / c1) + 1);

        a = ((i9 / c1) + 1);
    }
}
```

Division sets div-by-zero flag

Addition resets div-by-zero flag

Software state will affect **analog state**!

Hardware Trojans

Israeli sky-hack switched off Syrian radars countrywide

Backdoors penetrated without violence

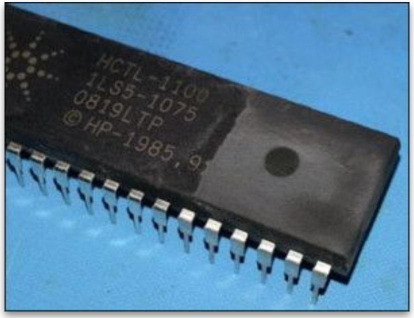
Lewis Page — Thu 22 Nov 2007 // 13:57 UTC

More rumours are starting to leak out regarding the mysterious Israeli air raid against Syria in September. It is now suggested that "computer to computer" techniques and "air-to-ground network penetration" took place.

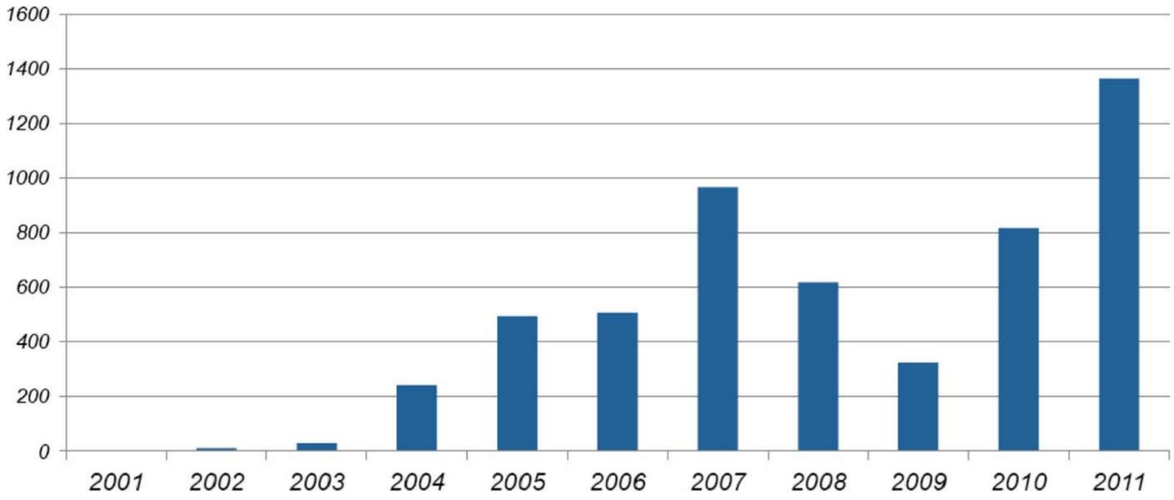
The latest revelations are made by well-connected *Aviation Week* journalists. Electronic-warfare correspondent David Fulghum says that US intelligence and military personnel "provided advice" to the Israelis regarding methods of breaking into the Syrian air-defence network.

Recycled and Counterfeit Hardware

Guin *et al.*: Counterfeit Integrated Circuits: A Rising Threat in the Global Semiconductor Supply Chain

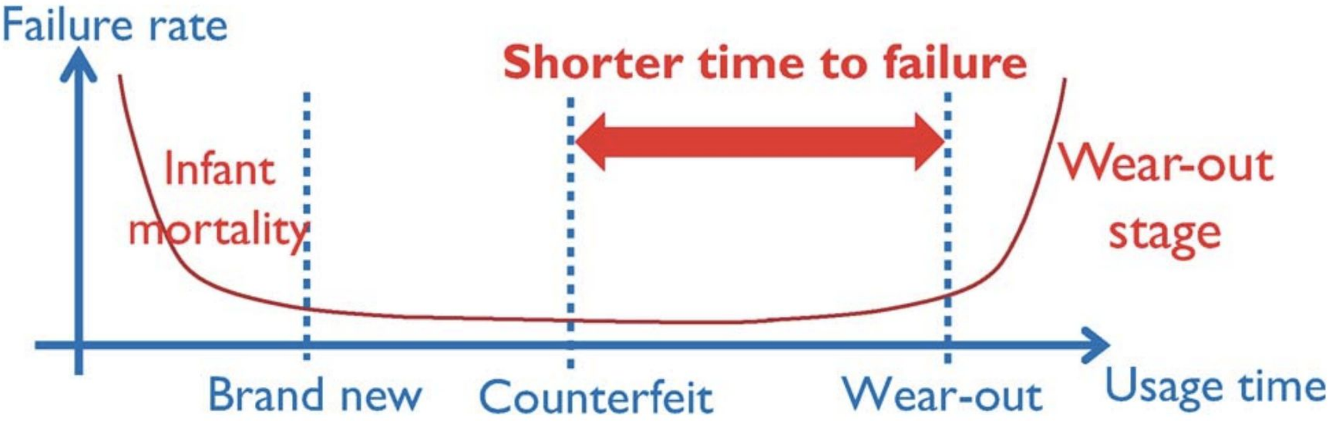



Russia is resorting to putting computer chips from dishwashers and refrigerators in tanks due to US sanctions, official says



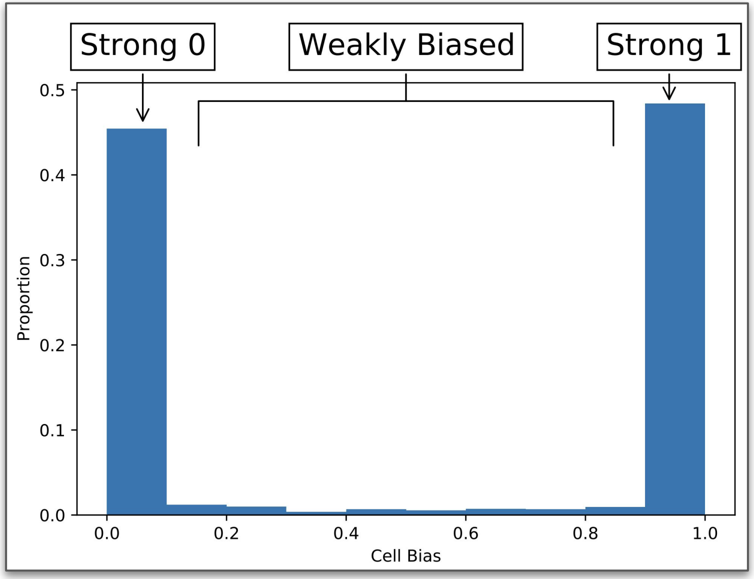
Recycled and Counterfeit Hardware

- **Counterfeit** and **recycled chips** have a **shorter lifespan**
  - Absolutely dangerous for security-critical use cases



Recycled and Counterfeit Hardware

- **Counterfeit** and **recycled chips** have a **shorter lifespan**
  - Absolutely dangerous for security-critical use cases



Secure Hardware

- **Can we ever know for sure that a chip is secure?**

Next time on CS 4440...

Election Security