What if

software had no bugs?

Herbert Bos

Vrije Universiteit Amsterdam


WHY WOULD I CARE?!

2010

Security problems are caused by

- **Software bugs**, and
- **Configuration bugs**

Since 2016

Even if the software is perfect

–and well-configured

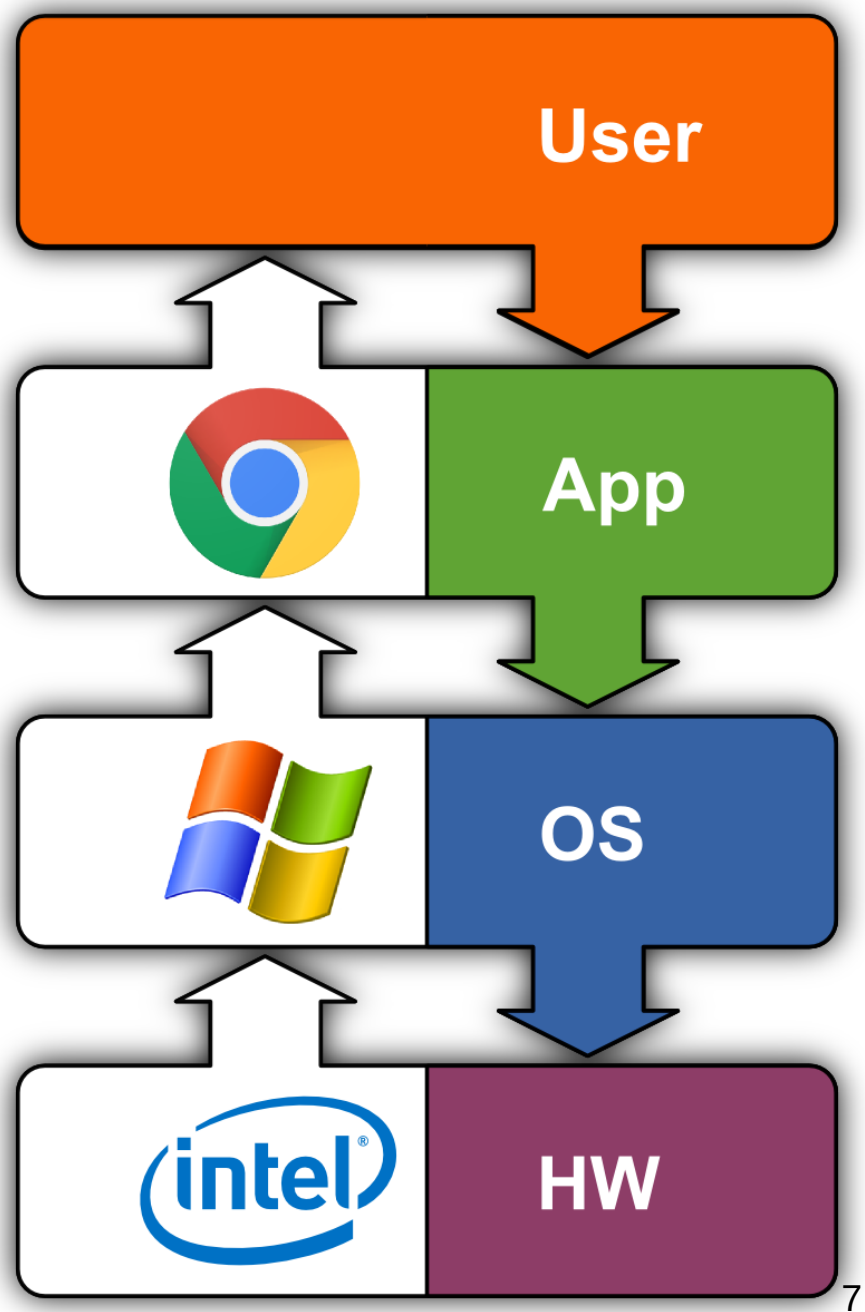
it is **still vulnerable!**

What does that mean for

formally verified systems?

Software Exploitation:

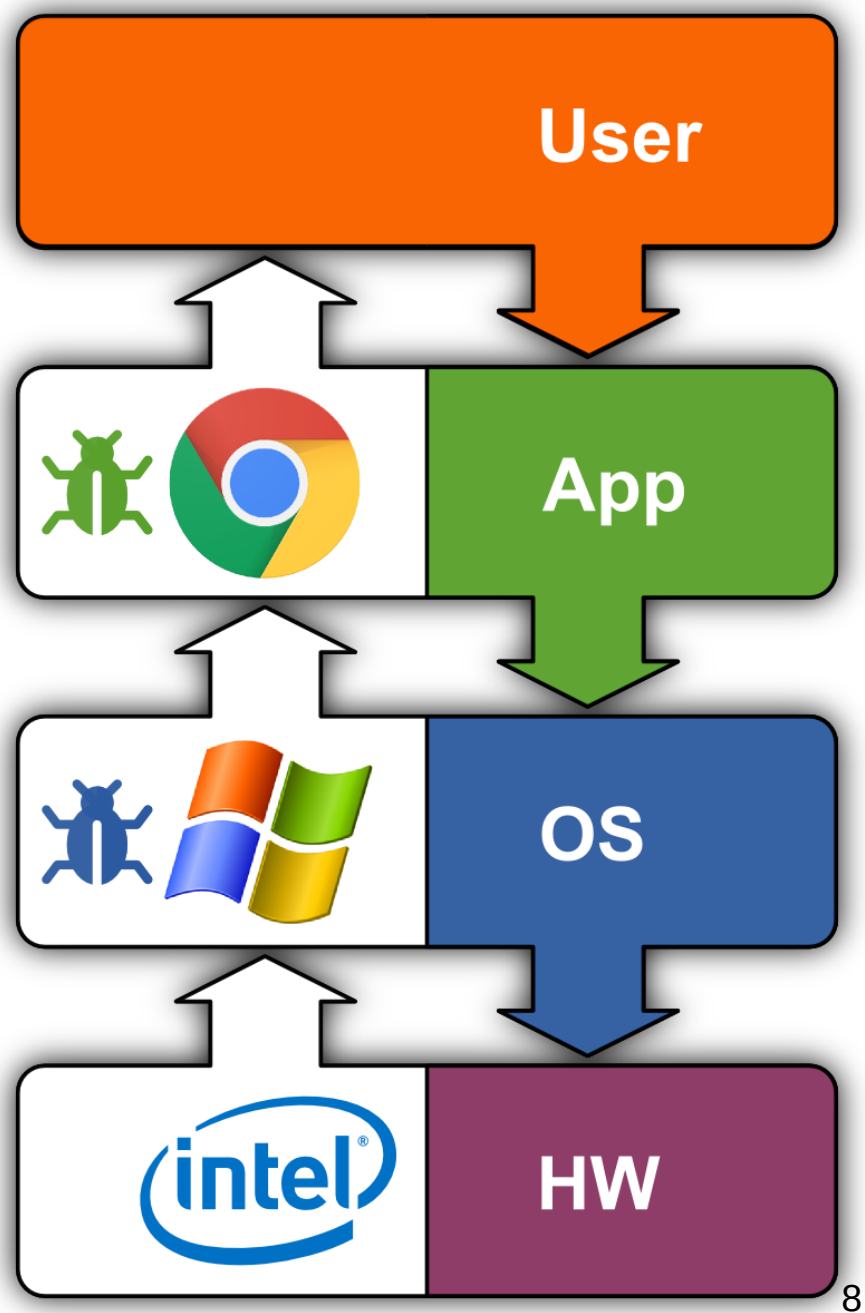
2010



Software Exploitation:

2010

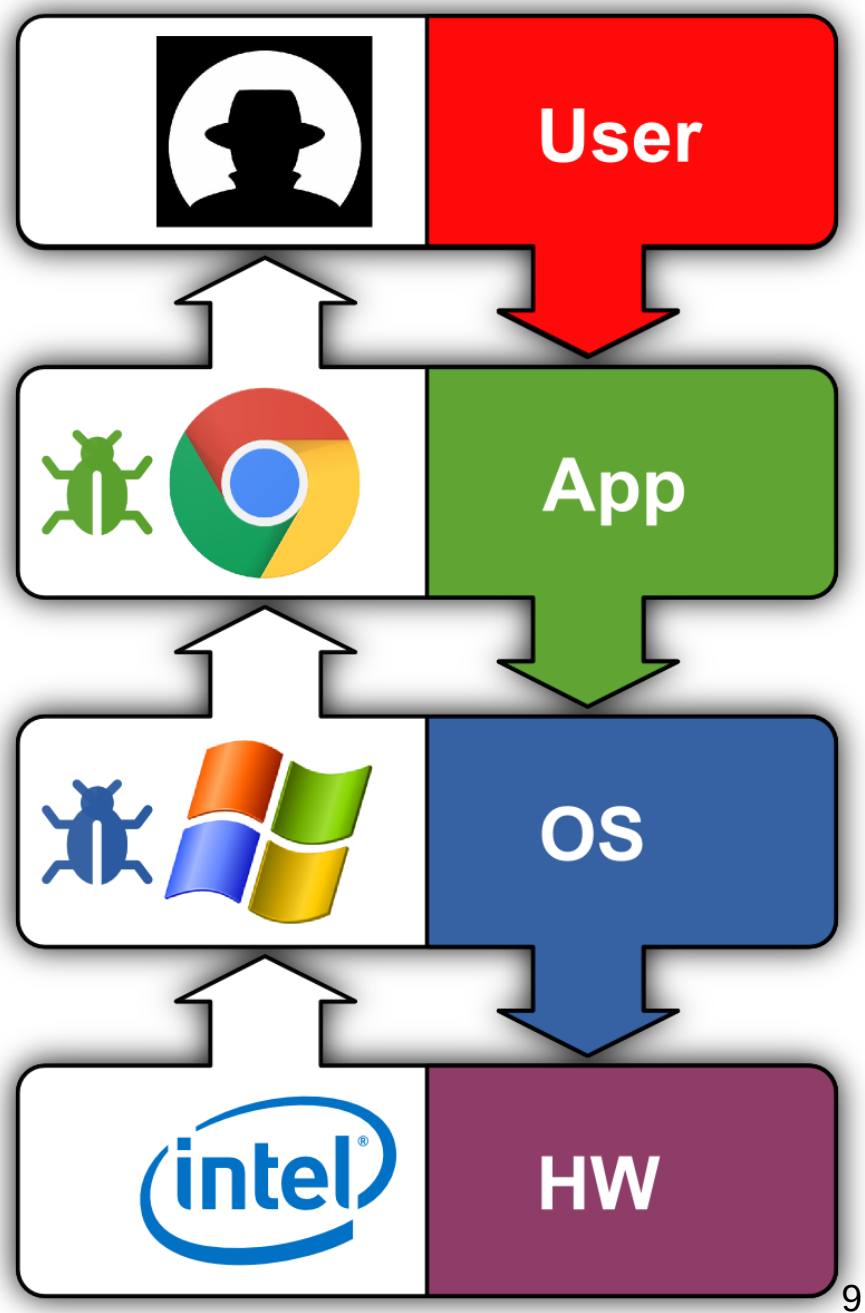
Bugs, Bugs Everywhere!



Software Exploitation:

2010

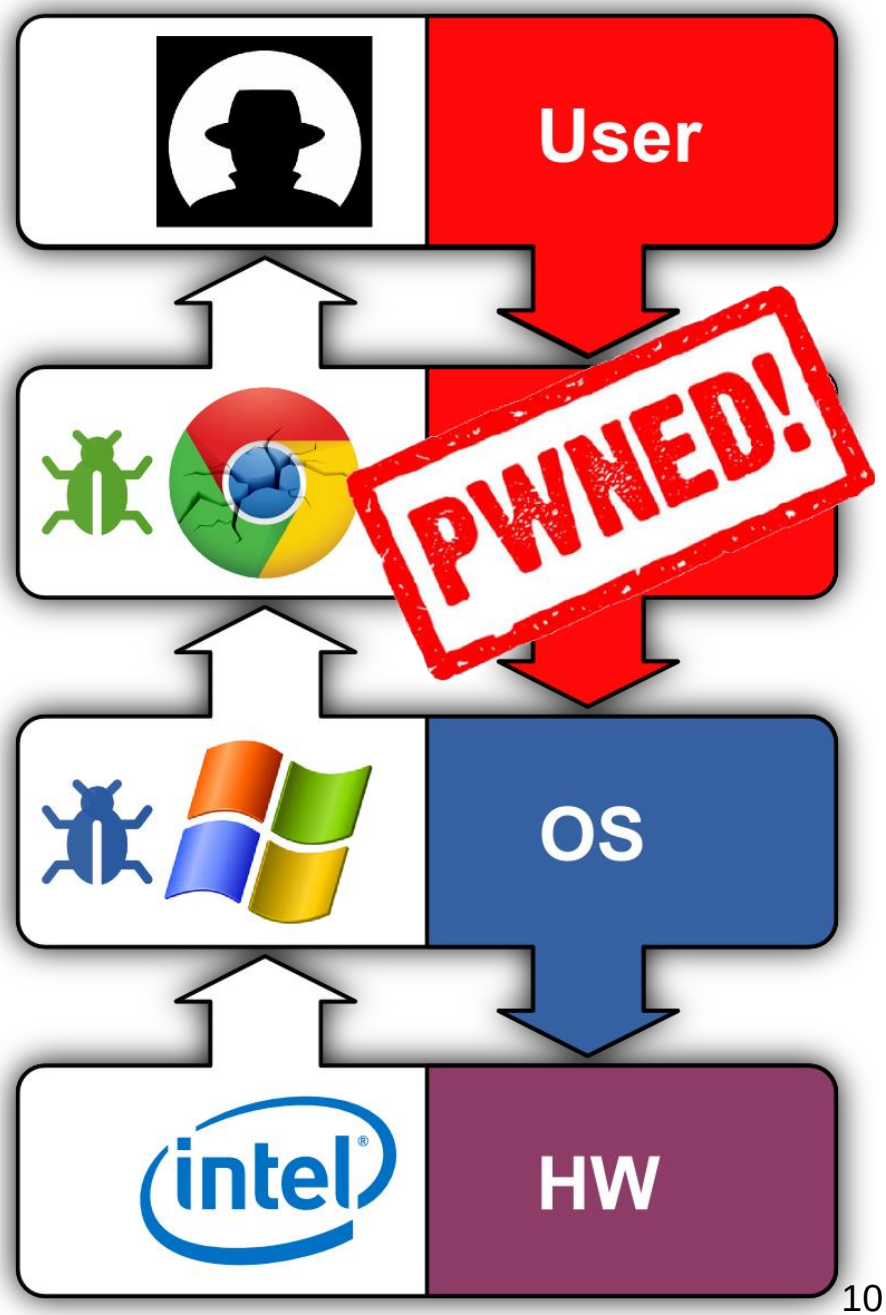
Attacker Exploits Vulnerable Software



Software Exploitation:

2010

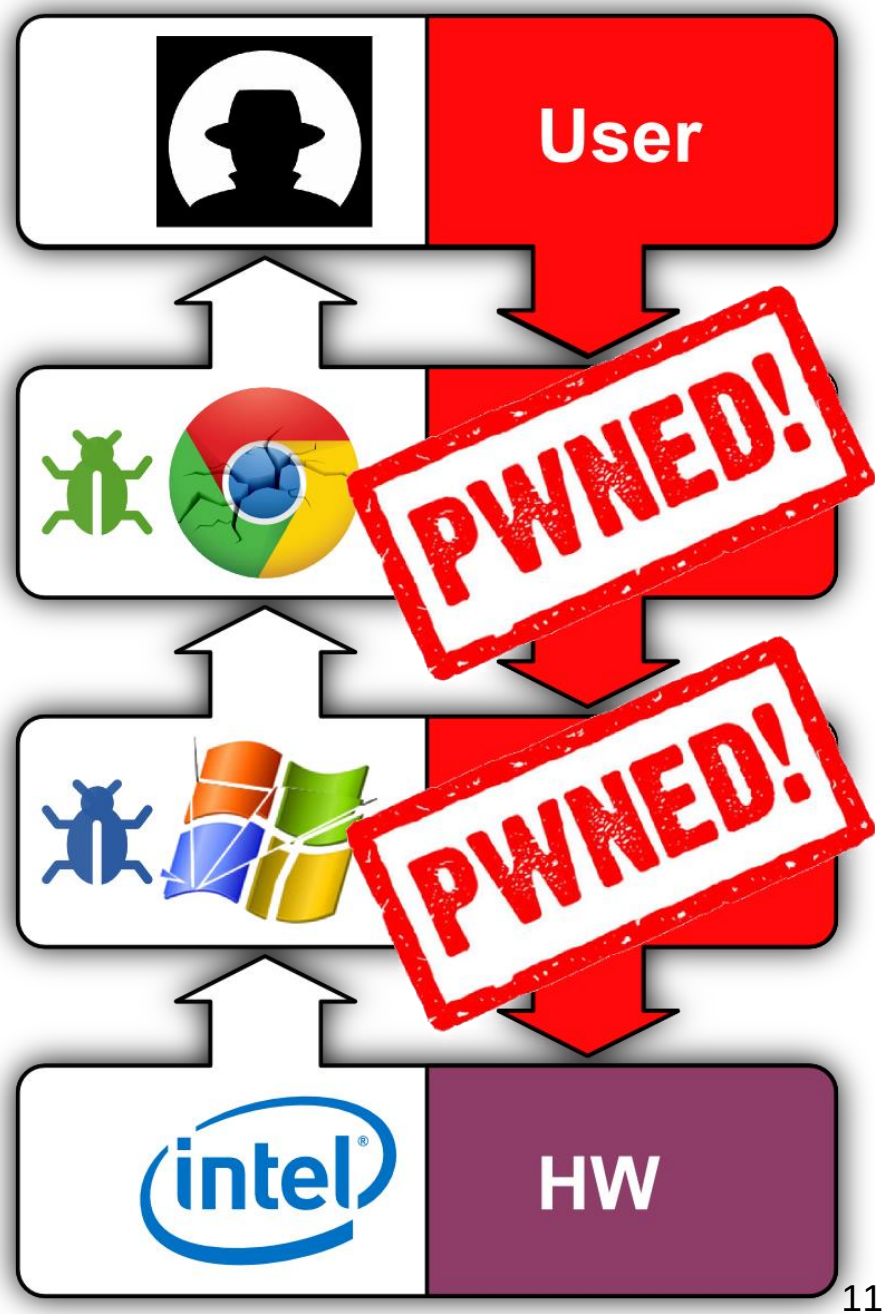
Attacker Owns Application



Software Exploitation:

2010

Attacker Owns System



Software Exploitation: 2010

Systems security problems caused by **bugs**

- Software and configuration bugs
- Weak security implementations

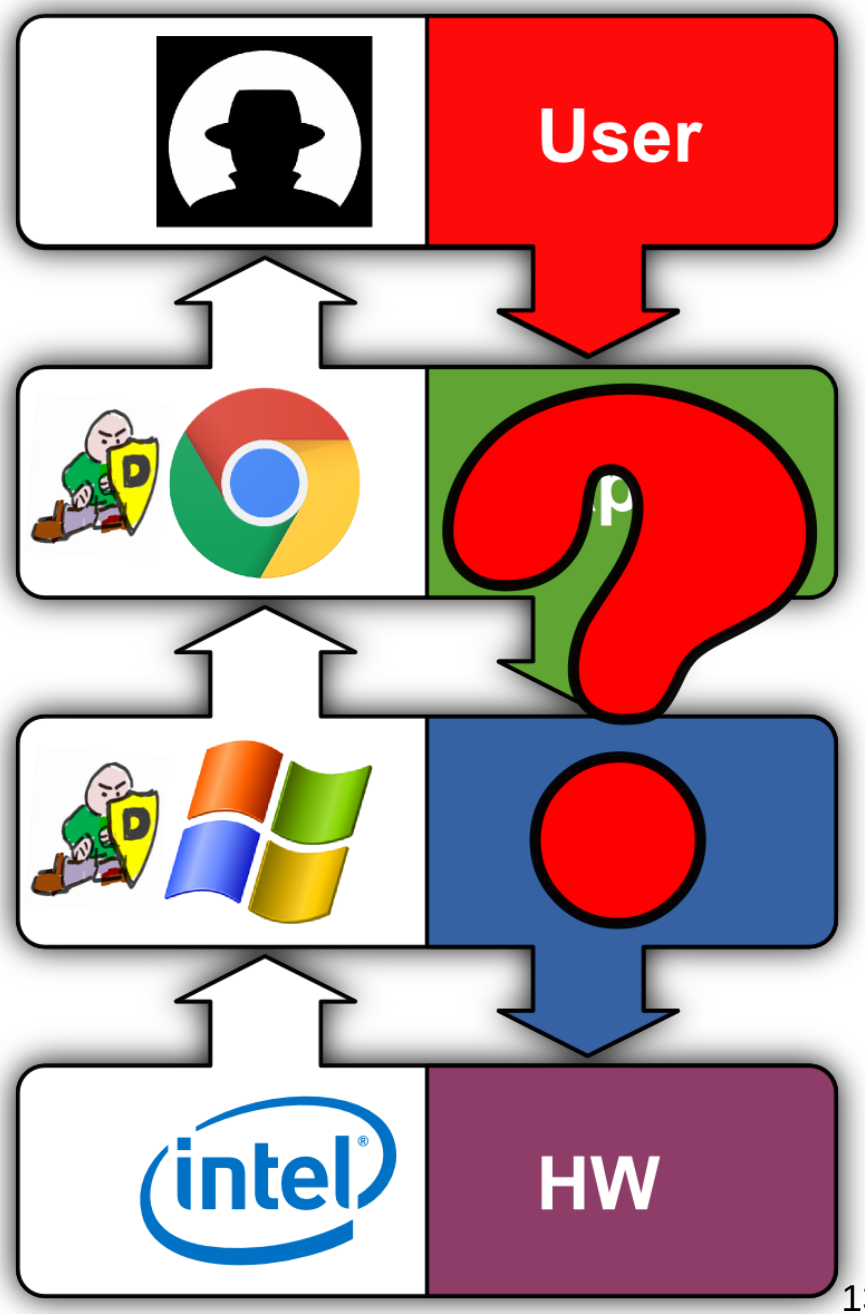
Impossible to write software without bugs

- However, we can mitigate their impact
- Many defenses proposed by industry and academia

Software Exploitation:

2016

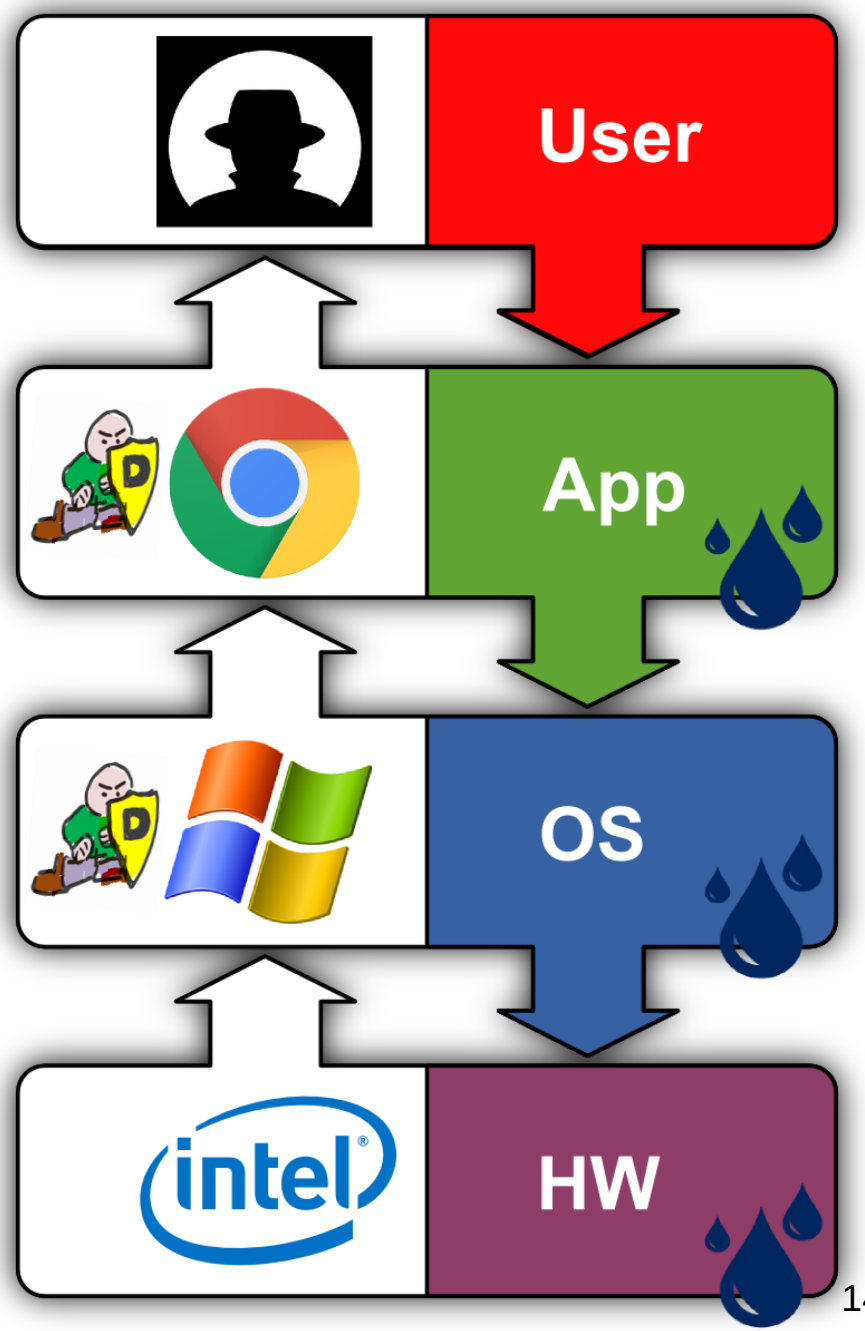
How to Find Memory R/W Primitives?



Software Exploitation:

2016

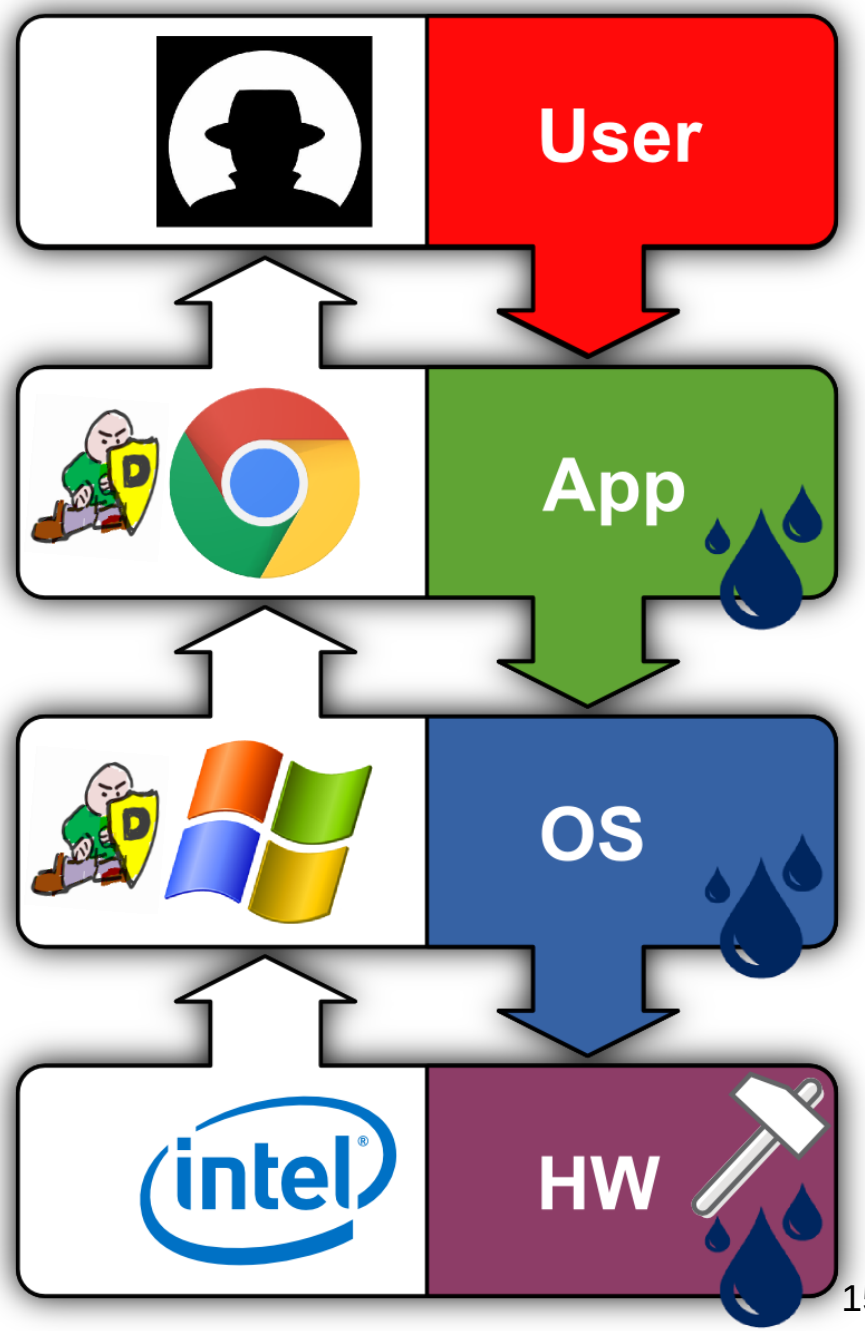
Memory R: Hw/Sw Side Channels



Software Exploitation:

2016

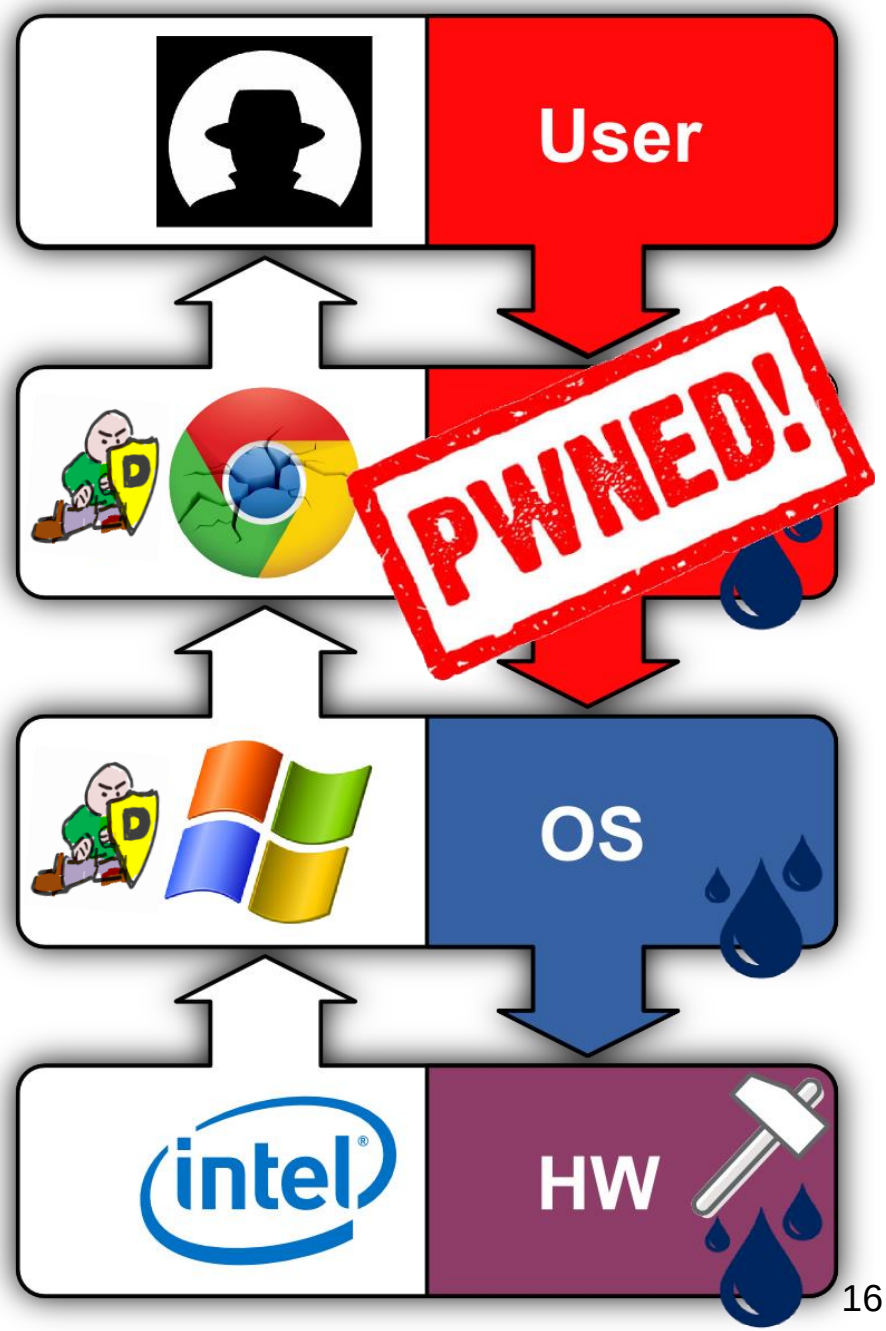
Memory W: Hardware Glitches



Software Exploitation:

2016

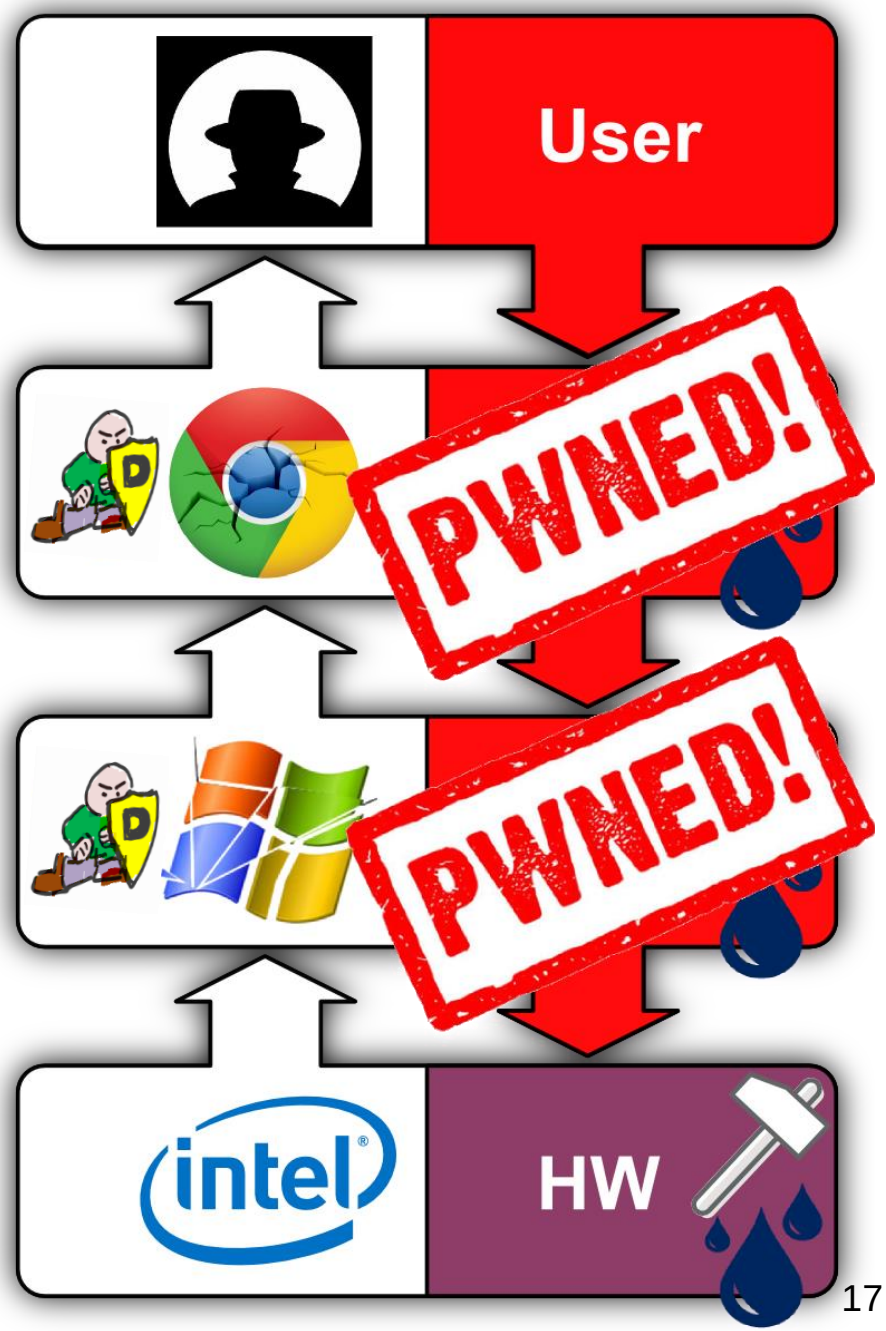
Memory R/W: Back to Reliable Exploits



Software Exploitation:

2016

Memory R/W: Back to Reliable Exploits

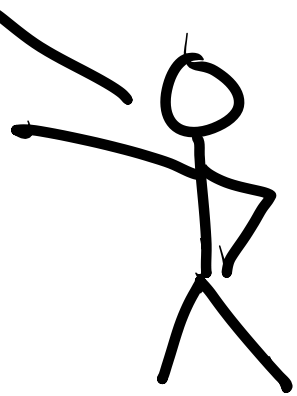


YES!

SO...

EVEN IF THE SOFTWARE IS PERFECT

WE ARE ALL STILL SCREWED?



Software Exploitation: 2016

Even if the software is **perfect**...

...with no bugs, well-configured, and latest defenses

...it is still **vulnerable**!

Attackers abuse **properties** of modern hw and sw for reliable exploitation

We'll look at **3 examples** (browsers, clouds) with **3 properties (**dedup, Rowhammer, speculation)

EXAMPLE 1

Meltdown/Spectre

Meltdown & Spectre

The Bugs That Shook The World

Herbert Bos

Vrije Universiteit Amsterdam

MICRO ARCHITECTURAL ATTACKS

JUST LIKE ROWHAMMER

CACHE SIDE CHANNELS

AND

GPU (GRAND PWNING UNIT)

AND MORE

BTW THE NAMING AND BRANDING IS SUPER CONFUSING

MELTDOWN ↔ SPECTRE

VARIANT 1, 2, 3

(WHERE 3 = MELTDOWN
AND 1+2 = SPECTRE)

CRAZY. WE WILL IGNORE THIS.

①

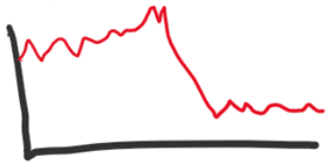
SO THERE WERE THESE TWO VULNERABILITIES

THEY EVEN HAVE BRAND NAMES: MELTDOWN & SPECTRE

AND THEIR OWN LOGOS

THAT APPEARED ON CNN

AND MADE INTEL'S STOCK GO:



SO... WHAT ARE THEY

AND WHY SHOULD I CARE?

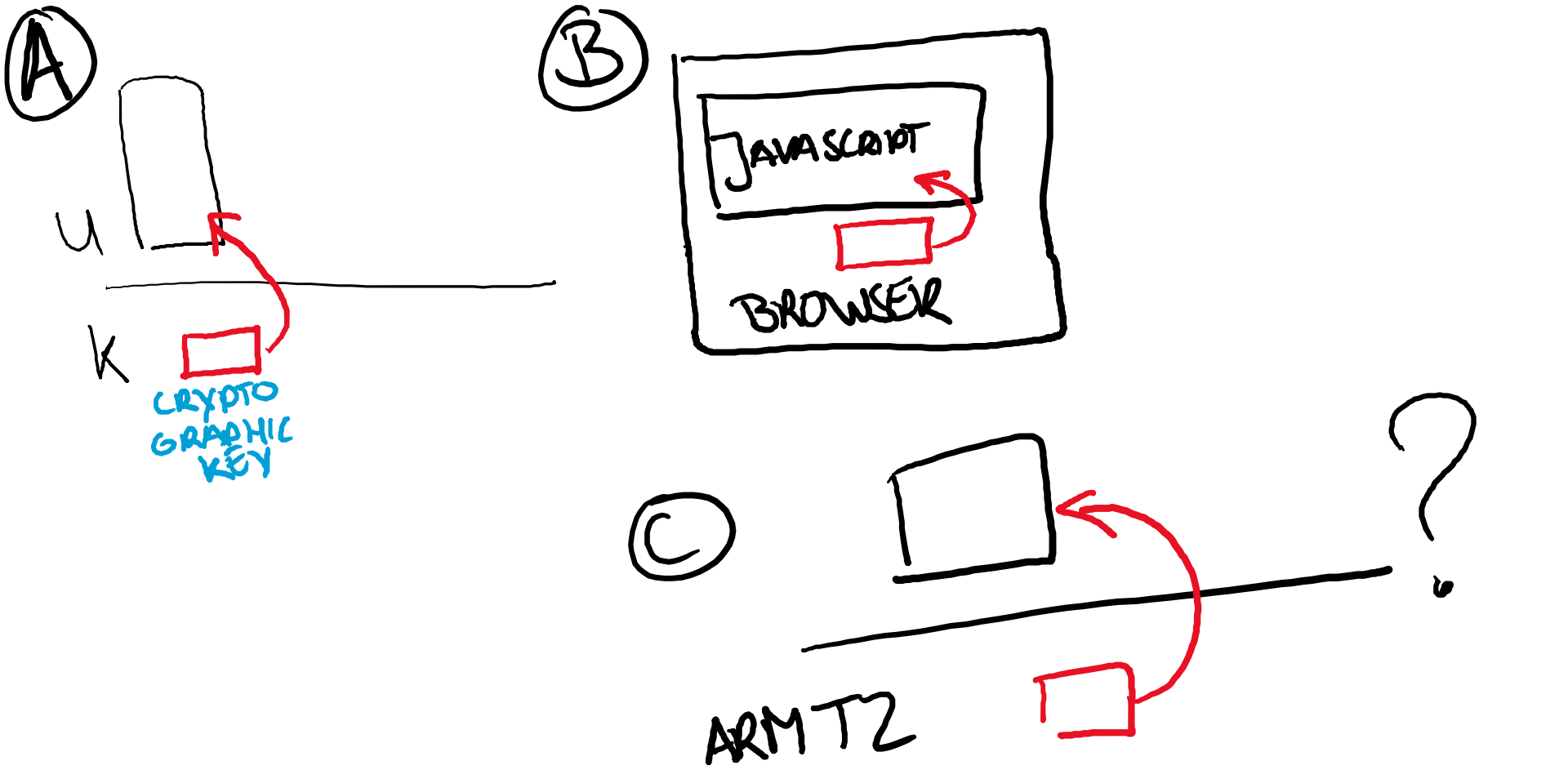
(2) WELL, THE PROBLEM IS THAT THEY ALLOW ATTACKERS TO ACCESS THE MOST PRIVILEGED DATA IN THE SYSTEM — IN THE KERNEL

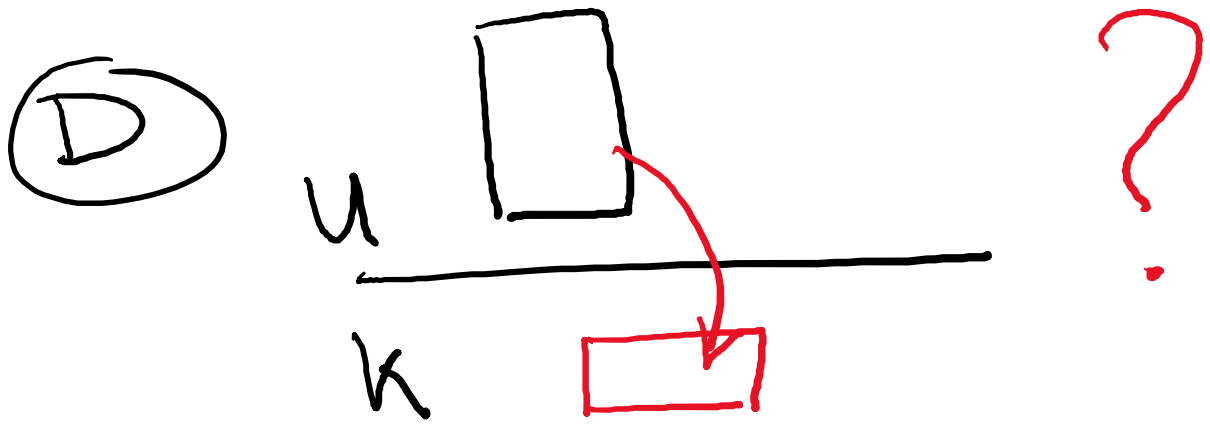
(3) THE KERNEL - USERSPACE BOUNDARY IS SACRED

IF NOT MAINTAINED ALL HELL BREAKS LOOSE

MOREOVER THESE VULNERABILITIES ARE IN THE HARDWARE 

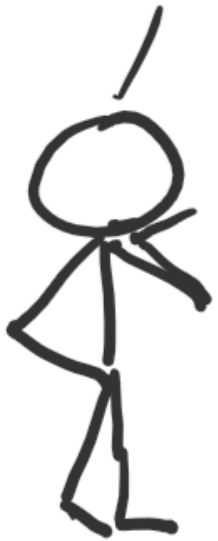
NOT EASY TO FIX!


A
U
K
CRYPTO
GRAPHIC
KEY
B
JAVASCRIPT
BROWSER
C
?
ARM T2


D
U
K
?

4

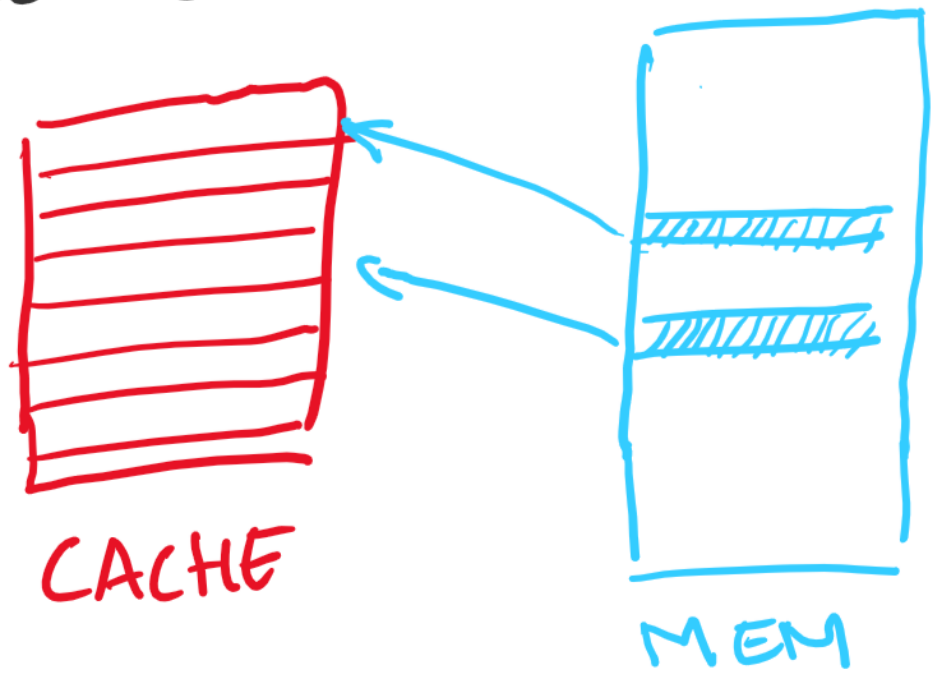
SO, WHAT'S UP WITH THE HARDWARE?

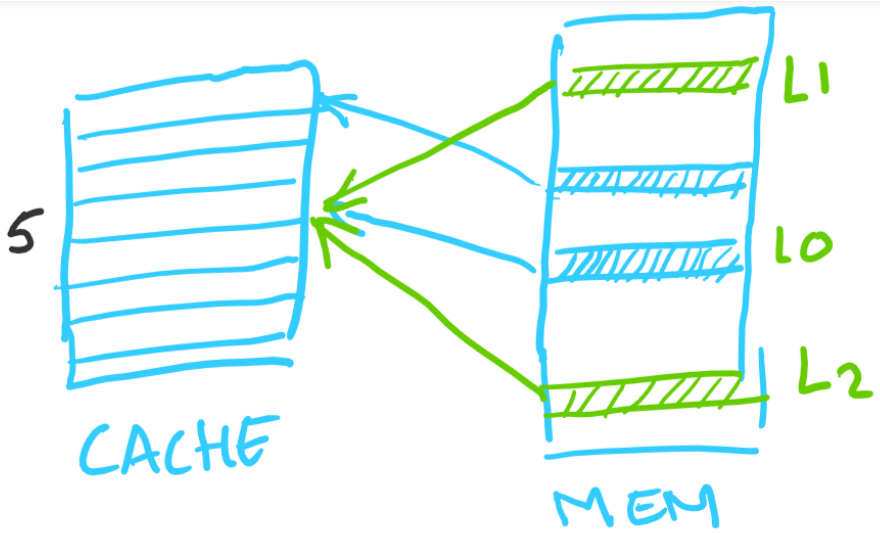


5 WORKS AS FOLLOWS

WHENEVER CPU ACCESSES DATA IN MEMORY, THIS DATA IS STORED IN THE CACHE FIRST → VERY FAST MEMORY

(6) DATA STAYS THERE UNTIL THE CACHEENTRY IS NEEDED FOR SOMETHING ELSE (THE CACHE IS VERY SMALL)





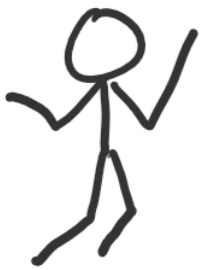
*ACTUALLY SLIGHT SIMPLIFICATION — REAL CACHES ARE "N-WAY SET-ASSOCIATIVE" BUT NOT IMPORTANT HERE

CACHE IS SMALL — SO MULTIPLE MEMORY LOCATION MAP ONTO THE SAME CACHE ENTRY

FOR INSTANCE: CACHE LINE 5 MAY FIRST CONTAIN THE DATA AT L0, THEN L_1 THEN L_2 (AND THEN PERHAPS L_1 AGAIN, ETC)

7

SO WHAT?



WAIT – BEAR WITH US

WE NEED TO EXPLAIN SOME OTHER THINGS FIRST

CACHES ARE NOT THE ONLY TRICK TO MAKE COMPUTERS FAST

9.A. OUT OF ORDER EXECUTION

SOME INSTRUCTIONS TAKE A LONG TIME. IF SUBSEQUENT INSTRUCTIONS DO NOT DEPEND ON RESULT, THEY MAY EXECUTE AND COMPLETE FIRST

X = Y * Z → SLOW

A = B + 1 → FAST

ALSO SOMETIMES CPU SPECULATES ON OUTCOME

WRITE 1 IN MEMORY LOCATION X

MUST CHECK IF THIS LOCATION IS ACCESSIBLE FIRST

Y = X + 1 → BUT TO SPEED THINGS UP CPU ALREADY CALCULATES THIS

→ IN CASE OF ERROR → SQUASH RESULTS!

9.B.

SPECULATIVE EXECUTION GOES VERY FAR

NORMALLY TRUE

```
IF (X == 42) {
    Y = 3;
    Z = Y * X;
    ...
}
```

⇒

```
IF (X == 42) {
    Y = 3;
    Z = Y * X;
    ...
}
```

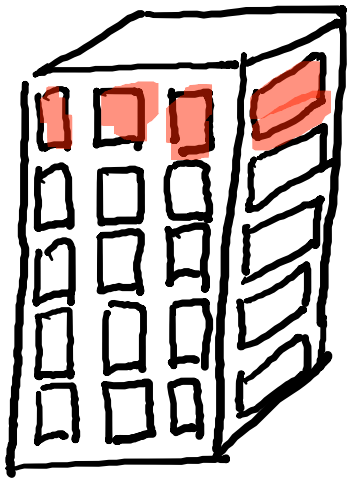
SPECULATIVELY EXECUTE THIS CODE WHILE CHECKING THE CONDITION

WHEN THE GUESS WAS CORRECT, WE IMMEDIATELY HAVE THE RESULT

OTHERWISE, SQUASH THESE RESULTS COMPLETELY

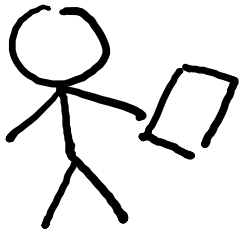
11

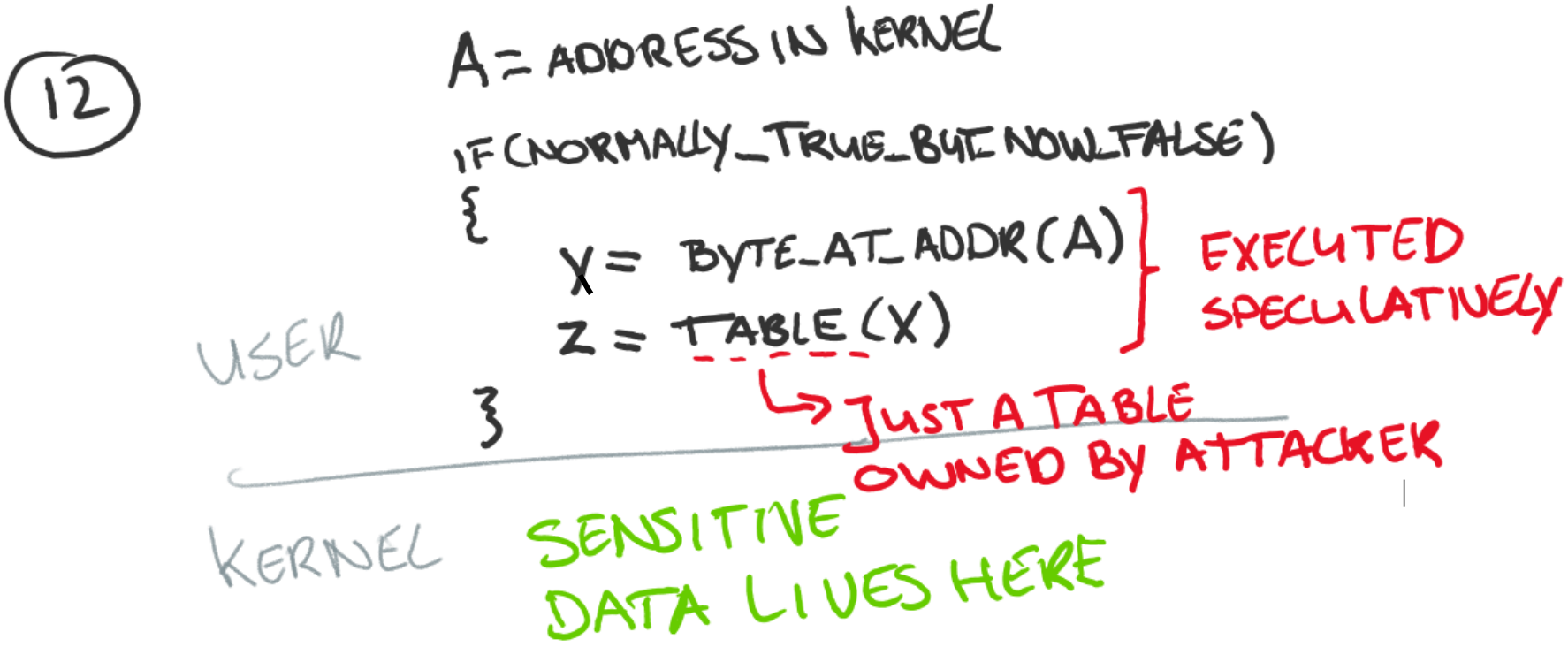
WELL HERE COMES THE TRICK

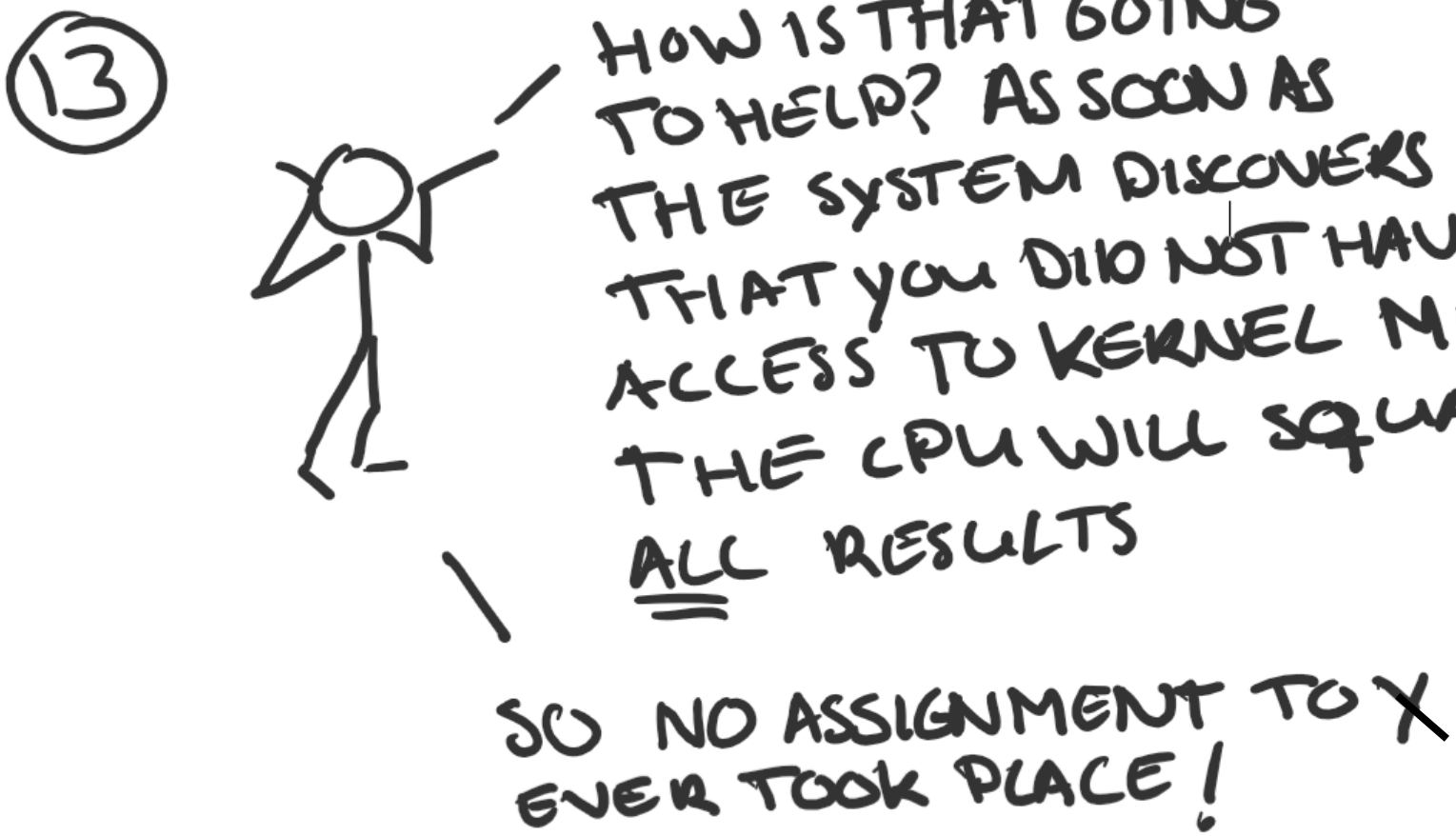


APPARTMENT # 13

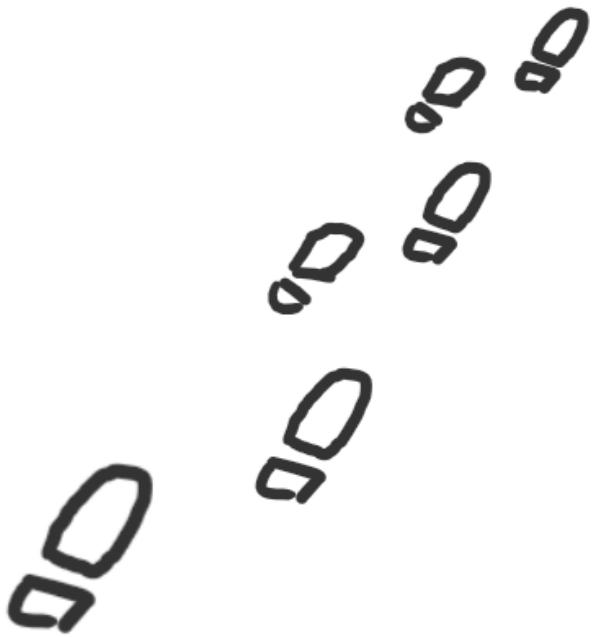
IS ALWAYS THE SAME



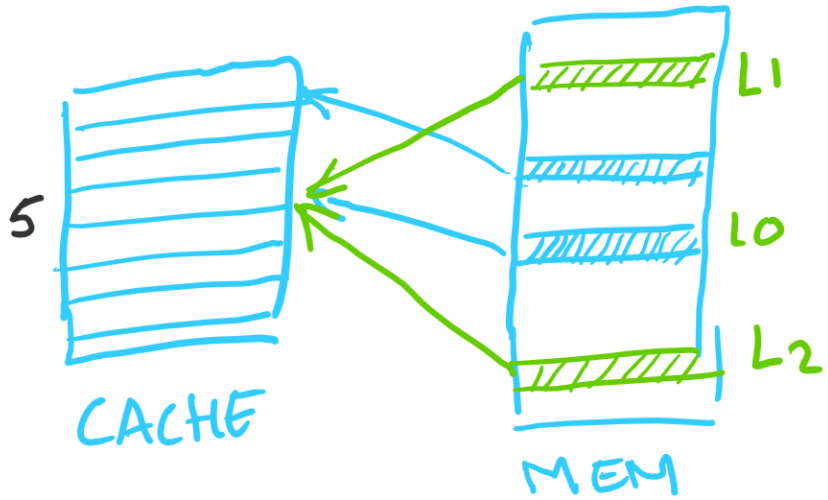

12
A = ADDRESS IN KERNEL
IF (NORMALLY_TRUE_BUT_NOW_FALSE)
{
X = BYTE_AT_ADDR(A)
Z = TABLE(X)
}
EXECUTED SPECULATIVELY
TABLE
JUST A TABLE OWNED BY ATTACKER
USER
KERNEL
SENSITIVE DATA LIVES HERE


13
HOW IS THAT GOING TO HELP? AS SOON AS THE SYSTEM DISCOVERS THAT YOU DID NOT HAVE ACCESS TO KERNEL MEM, THE CPU WILL SQUASH ALL RESULTS
SO NO ASSIGNMENT TO X EVER TOOK PLACE!

14 THIS IS TRUE.
BUT THERE IS
STILL A TRACE!



(15) RECALL THE CACHE

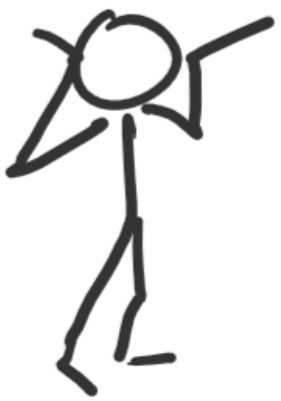



```
A = ADDRESS IN KERNEL
IF (NORMALLY_TRUE_BUT NOW_FALSE)
{
    X = BYTE_AT_ADDR(A)
    Z = TABLE(X)
}
```

↓

WHEN CPU SPECULATIVELY EXECUTES THIS CODE, IT WILL LOAD THE DATA IN THIS TABLE ENTRY IN THE CACHE.

16



WHAT GOOD DOES THAT DO? ALL RESULTS WILL BE SQUASHED

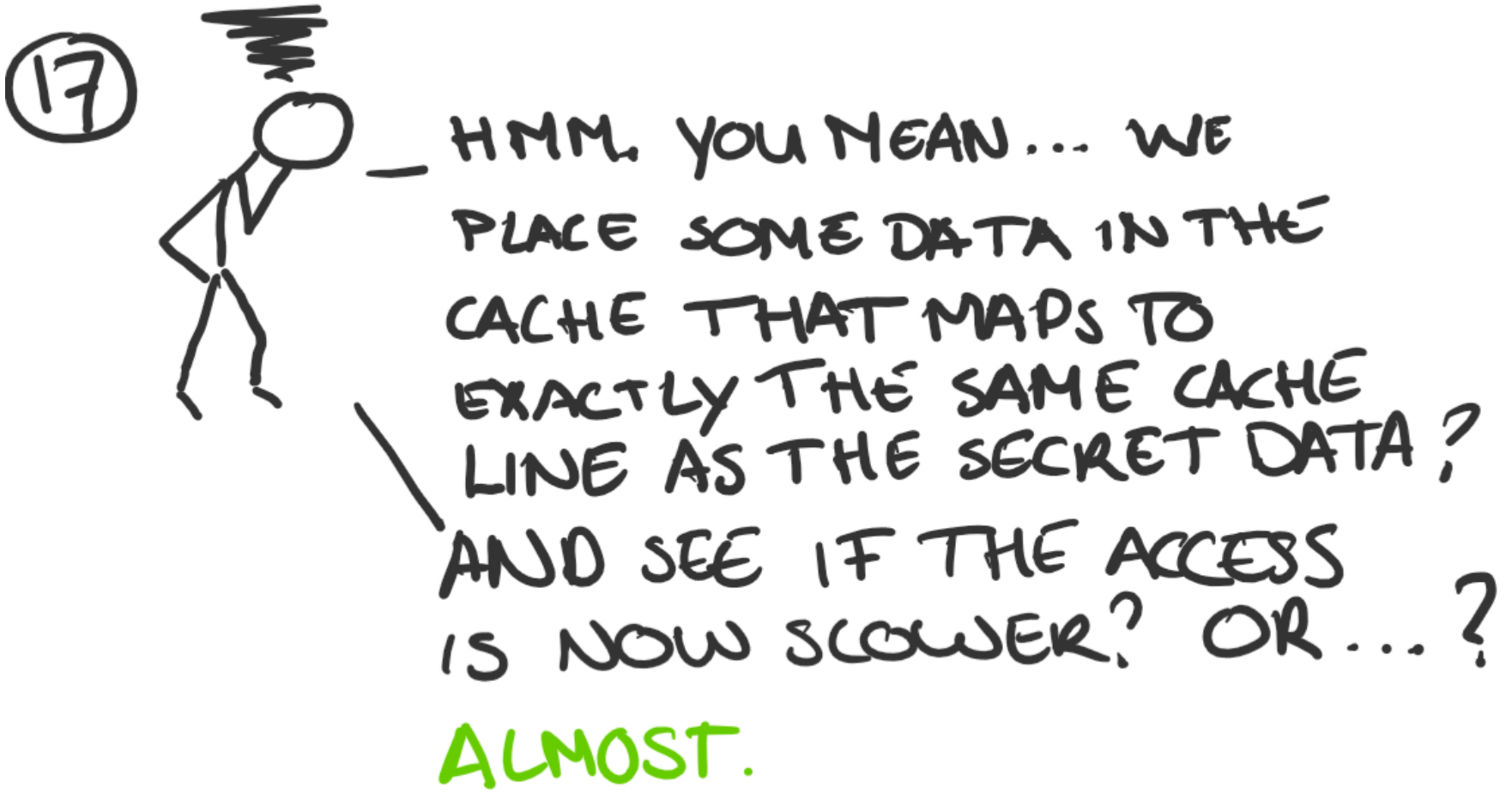
TRUE. BUT THE DATA IS STILL IN THE CACHE. UNTIL IT IS REPLACED

BUT THAT DOES NOT MATTER. NOBODY CAN ACCESS IT?!

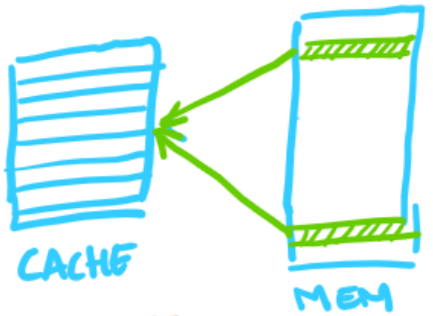
AAH. BUT ACCESSES TO THAT DATA WILL NOW BE MUCH FASTER.

IF WE COULD ACCESS IT, YES. BUT WE CAN'T. THAT WAS THE WHOLE POINT.

RIGHT. BUT MAYBE WE CAN MAKE THE EFFECT NOTICEABLE IN THE PERFORMANCE OF OUR OWN CODE!


17
HMM. YOU MEAN... WE PLACE SOME DATA IN THE CACHE THAT MAPS TO EXACTLY THE SAME CACHE LINE AS THE SECRET DATA?
AND SEE IF THE ACCESS IS NOW SLOWER? OR...?
ALMOST.

(18) FIRST, WE MAKE SURE THAT THE TABLE IS NOT IN THE CACHE (BY ACCESSING OTHER DATA THAT MAPS ON THE SAME CACHE LINES



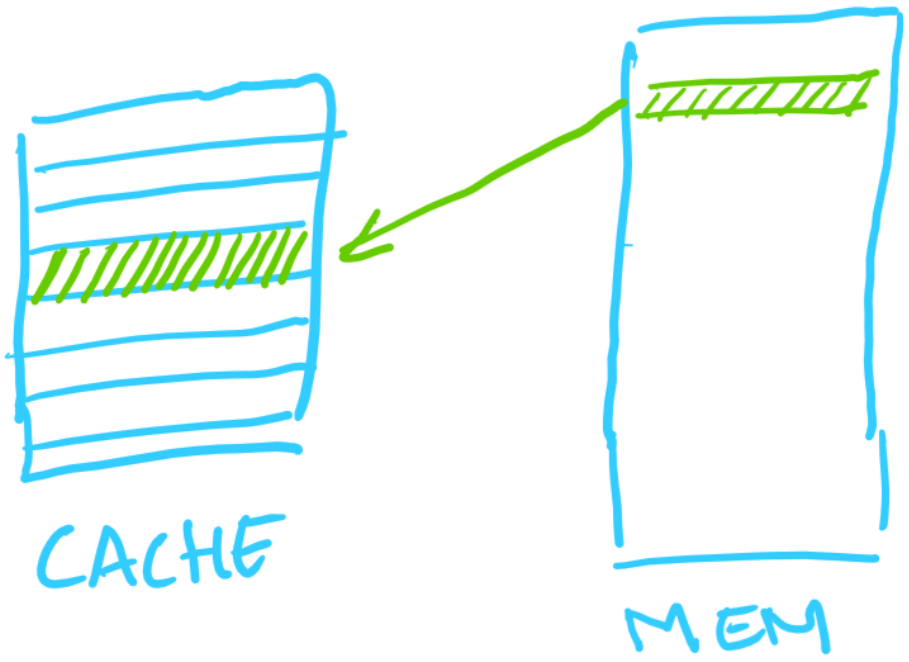
IF WE WERE TO EXECUTE THESE TABLE ENTRIES NOW (WHICH WE WON'T) — THIS WOULD TAKE NOTICEABLY LONGER THAN IF THEY HAD BEEN IN THE CACHE

NOW,
IMAGINE WE RUN THE FOLLOWING CODE — WHILE INITIALLY THE TABLE IS NOT IN THE CACHE

```
A = ADDRESS IN KERNEL
IF (NORMALLY_TRUE_BUT_NOW_FALSE)
{
    X = BYTE_AT_ADDR(A)
    Z = TABLE(X)
}
```

CACHE → EMPTY

WHAT WILL THIS DO?

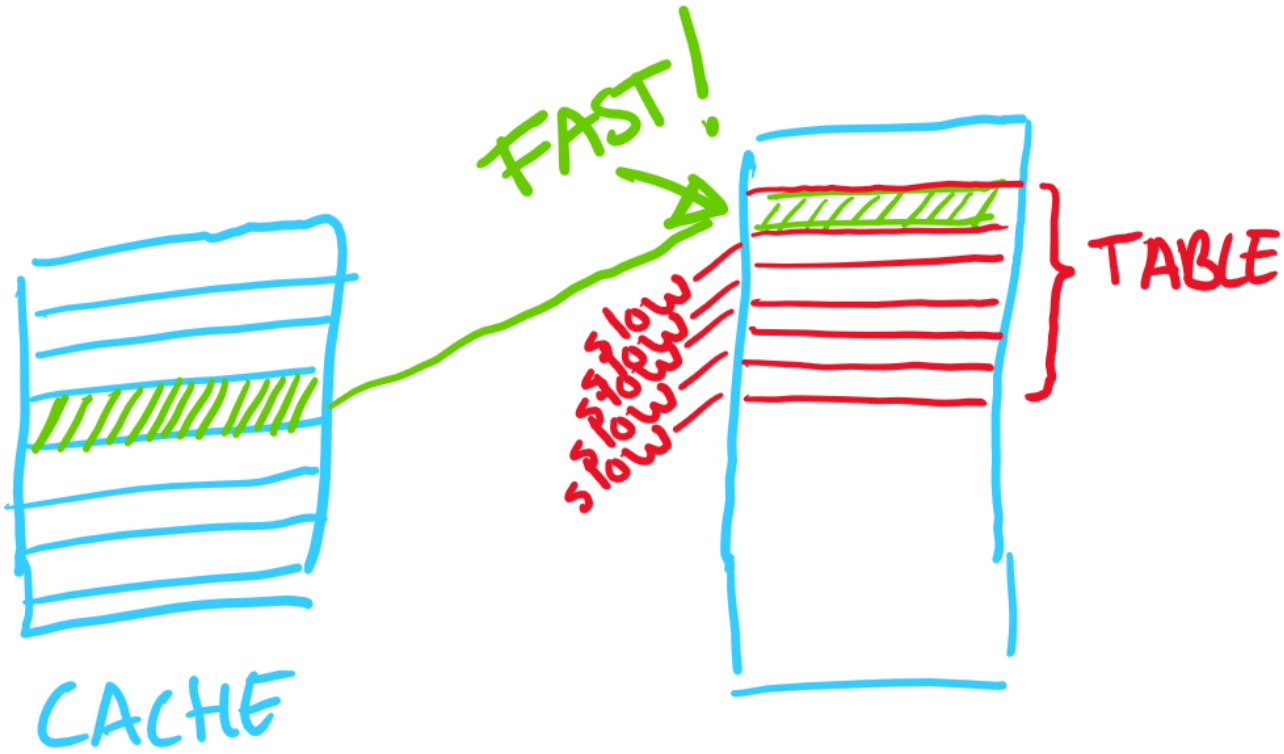


IT WILL LOAD THE VALUE AT TABLE(X) FROM MEMORY INTO THE CACHE.

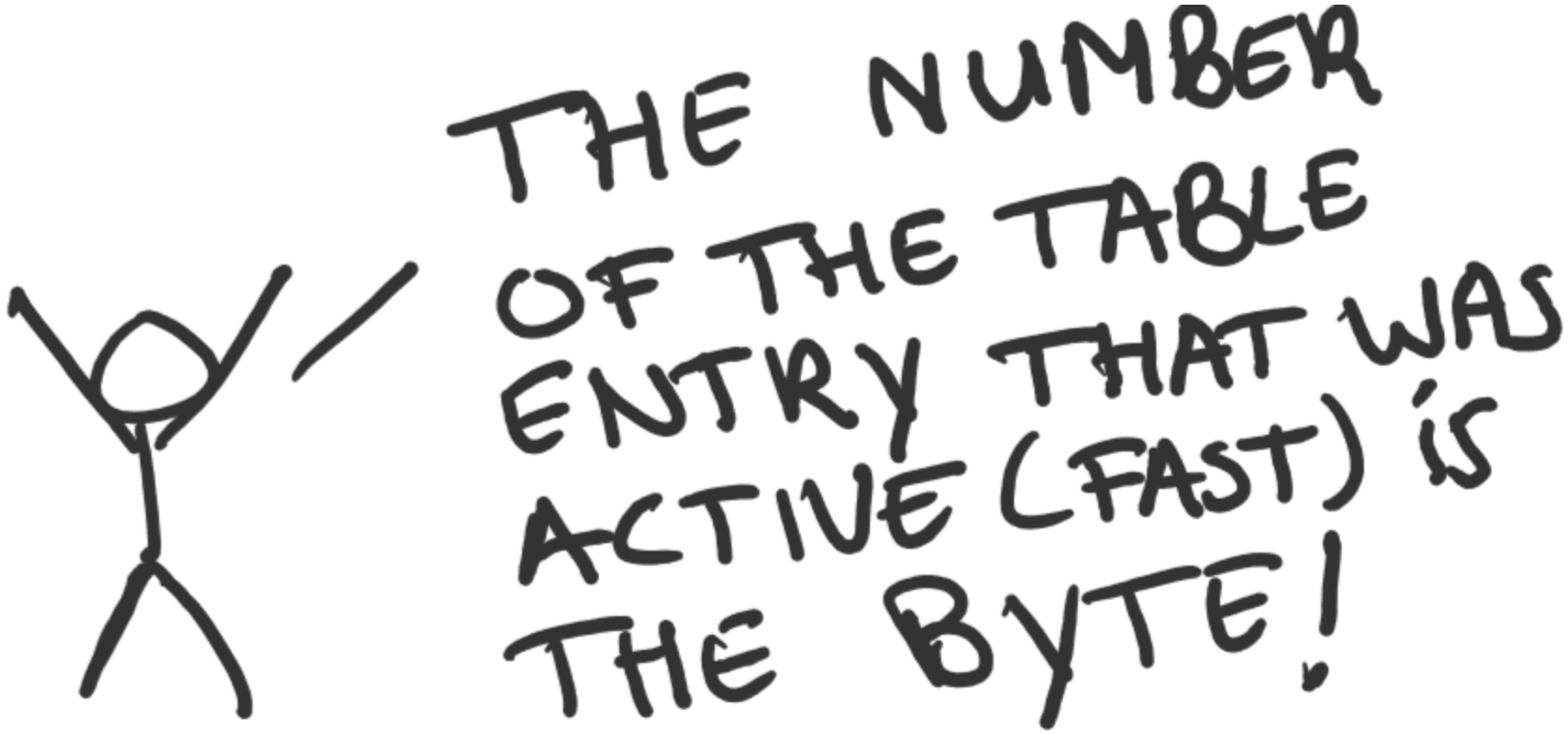
OF COURSE, THE ASSIGNMENT TO y IS SQUASHED (AS THE SPECULATIVE EXECUTION WAS NOT VALID)

HOWEVER, THE DATA IS STILL IN THE CACHE!

RIGHT. BUT WHAT HAPPENS IF WE ACCESS ALL ELEMENTS IN THE TABLE?



SO WE TIME THE ACCESSES
IF (<FAST) → THIS DATA WAS
ACCESSED BEFORE


THE NUMBER OF THE TABLE ENTRY THAT WAS ACTIVE (FAST) IS THE BYTE!

WORSE STILL:

THIS IS NOT JUST FOR THE KERNEL BUT FOR ALL SECURITY BOUNDARY WHERE CODE SHARES THE SAME ADDRESS SPACE

→ JAVASCRIPT IN A BROWSER!


20
SO HOW ARE WE GOING TO SOLVE THIS MESS?
DON'T LOOK AT ME!

(21) ACTUALLY, THE SOLUTION IS FAIRLY SIMPLE (BUT NOT CHEAP)

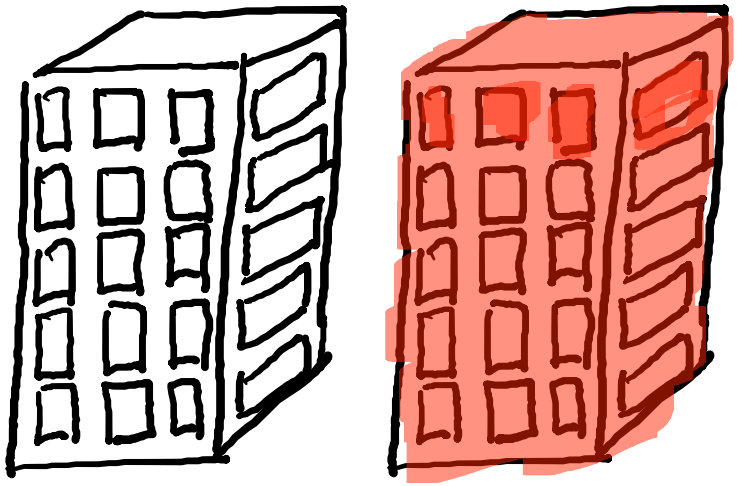
WELL FOR KERNEL

→ KNOWN AS KPTI (KERNEL PAGE TABLE ISOLATION)

⇒ JUST ENSURE THAT KERNEL AND USER PROCESS HAVE COMPLETELY SEPARATE ADDRESS SPACES

```
A = ADDRESS IN KERNEL
IF (NORMALLY_TRUE_BUT_NOW_FALSE)
{
    y = BYTE_AT_ADDR(A)
    z = TABLE(X)
}
```

CAN NO LONGER READ THIS



APPARTMENT # 13

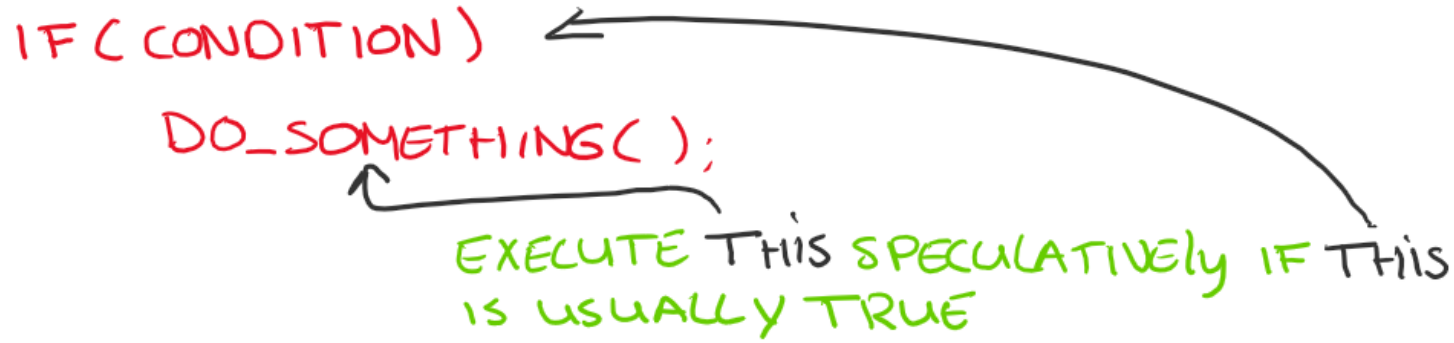
IS NOT THE SAME

22 UNFORTUNATELY, THIS IS NOT THE END OF IT

— THERE IS STILL MORE SPECTRE

— THAT PROBLEM GOES WAY DEEPER

23 REMEMBER WHAT WE SAID ABOUT SPECULATIVE EXECUTION



BUT HOW DOES CPU KNOW THAT IT IS USUALLY TRUE?

24 BRANCH PREDICTION

"I PREDICT THAT THE WEATHER TOMORROW WILL BE THE SAME AS TODAY"

FAIRLY ACCURATE WEATHER FORECAST

(25) BRANCH PREDICTION

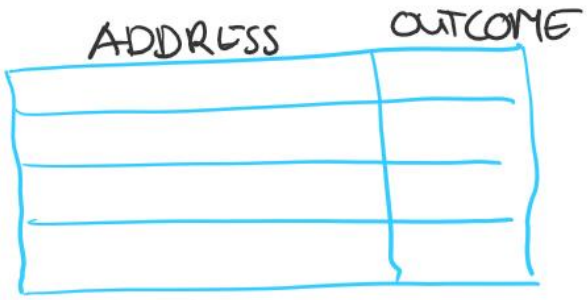
- CPU DOES SOMETHING SIMILAR
- IT TRACKS THE LAST N OUTCOMES FOR EACH BRANCH

```
IF (X==0)
...
ELSE
...
```

IT WILL PREDICT THE OUTCOME TO BE THE SAME AS THE LAST FEW OUTCOMES

AND STARTS SPECULATIVELY EXECUTING THE CORRESPONDING CODE

(26) SO HOW DOES IT STORE THIS?

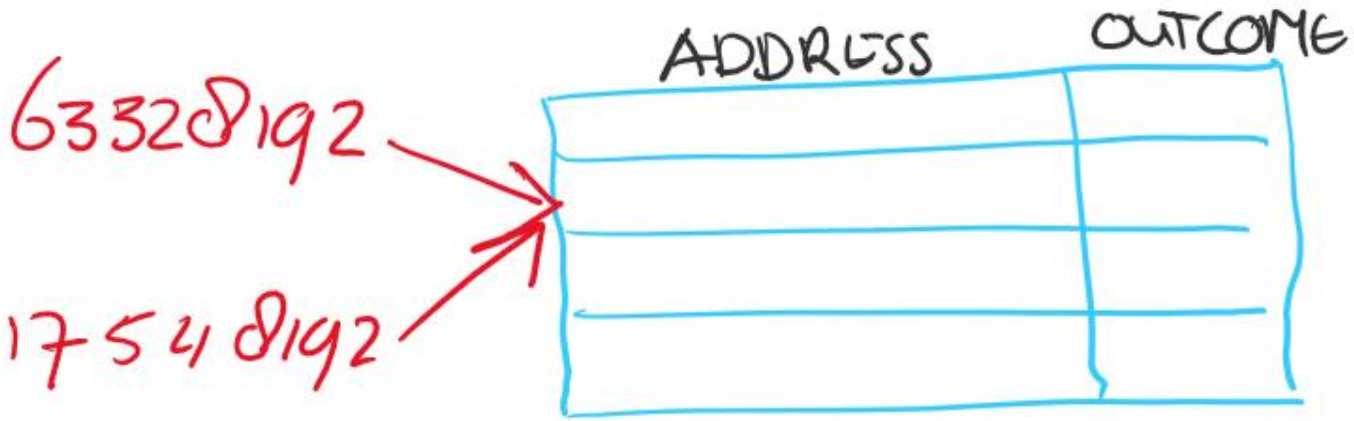


↳ BUT RATHER THAN STORING FULL ADDRESS (EXPENSIVE), IT WILL STORE PART OF THE ADDRESS

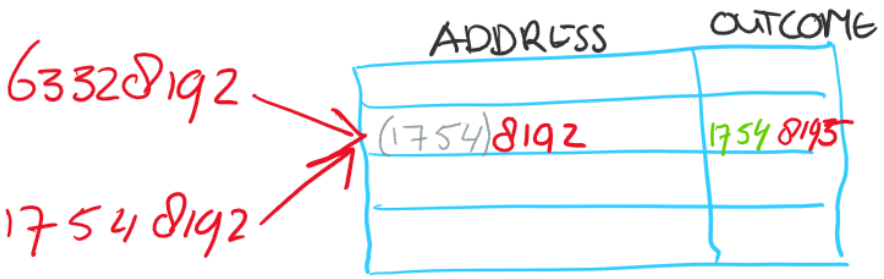
EXAMPLE: 63328192 → 8192

MUCH SHORTER AND PROBABLY UNIQUE

(27) IT DOES MEAN THAT WE (AGAIN) HAVE THE PROBLEM THAT MULTIPLE ADDRESSES (OF BRANCHES) MAP ON THE SAME ENTRY IN THE BRANCH PREDICTOR TABLE.



SAY THE BRANCH PREDICTOR TABLE LOOKS AS FOLLOWS

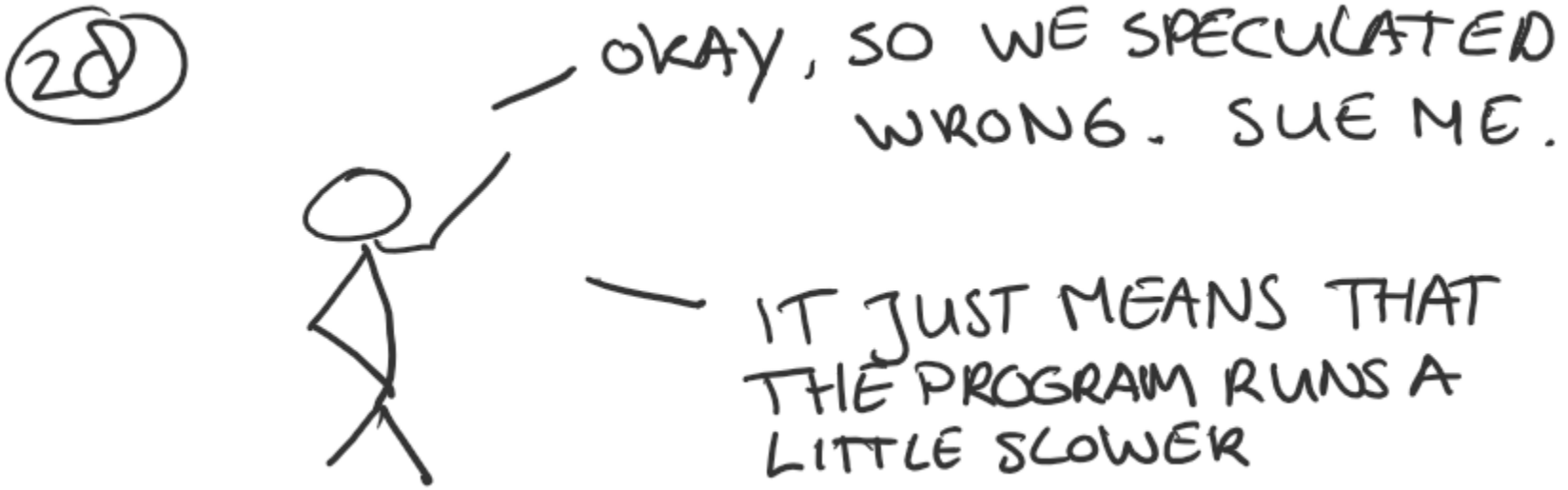


AND WE EXECUTE THE FOLLOWING:

6332 8192: IF (X == 0)

INSTRUCTION
INSTRUCTION
⋮

THE BP WILL TAKE THE ENTRY FOR THE OTHER BRANCH AS THE PREDICTION FOR THIS BRANCH


28
OKAY, SO WE SPECULATED WRONG. SUE ME.
IT JUST MEANS THAT THE PROGRAM RUNS A LITTLE SLOWER

(29) WRONG. IT MEANS THAT WE GIVE ANOTHER PROGRAM CONTROL OVER WHAT CODE WE (SPECULATIVELY) EXECUTE.

IF THE ATTACKER CREATES A BUNCH OF BRANCHES WITH THE SAME MAPPING AS THE BRANCHES IN THE VICTIM CODE AND TRAINS THE BRANCH PREDICTOR TO LEARN THAT THE BRANCHES ARE ALWAYS TAKEN (OR NOT TAKEN)

→ THIS OUTCOME WILL BE USED IN THE SPECULATIVE EXECUTION OF THE VICTIM CODE

→ ATTACKER CONTROLS VICTIM

→ CAN READ DATA IN SAME WAY AS BEFORE

30

SO, FOR INSTANCE, WE CAN LET THE KERNEL (SAY) JUMP TO CODE THAT LOADS SOME DATA IN THE CACHE

AND THEN USE THE CACHE SIDE CHANNEL TO LEAK THIS INFO AGAIN, YES

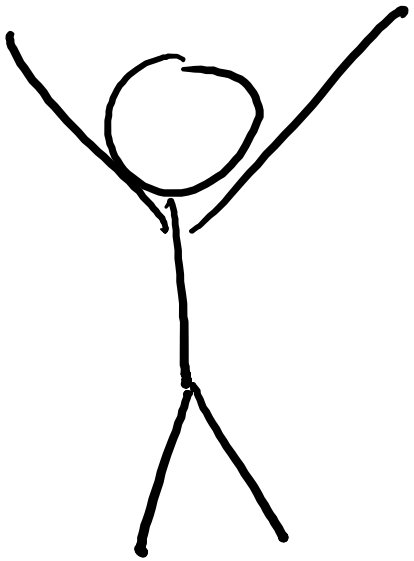
CONSIDER CODE LIKE

JMP *RAX → ALSO PREDICTED

RETURN → " "

GOT IT

HURRAY!



(31) HOW TO MITIGATE?

HARD. THERE ARE DIFFERENT VARIANTS.

WE CAN PLUG SOME HOLES — BUT ALL OF THEM?

JURY IS STILL OUT

(31) HOW TO MITIGATE ?

HARD. THERE ARE DIFFERENT VARIANTS.

WE CAN PLUG SOME HOLES — BUT ALL OF THEM?

JURY IS STILL OUT

skip

(32) TRICK #1 : ADD A SPECIAL LFENCE INSTRUCTION AFTER EVERY BOUNDS CHECK (OR SIMILAR)

→ STOPS SPECULATIVE EXECUTION ACROSS THE BARRIER

→ BRANCHES CONTROLLABLE BY ATTACKER

```
IF ( X IN THE RIGHT MEMORY AREA)
{
    LFENCE
    INSTR1
    INSTR2
    INSTR3
     :
}
```

→ NO SPECULATIVE EXECUTION

STOPS 1 VARIANT BUT DOES NOT STOP THE POLLUTION OF THE BRANCH PRED. FOR INSTANCE

(32) THE LFENCE INSTRUCTION CAN SIMPLY BE EMITTED BY COMPILER

→ RECOMPILE KERNEL AND YOU ARE DONE

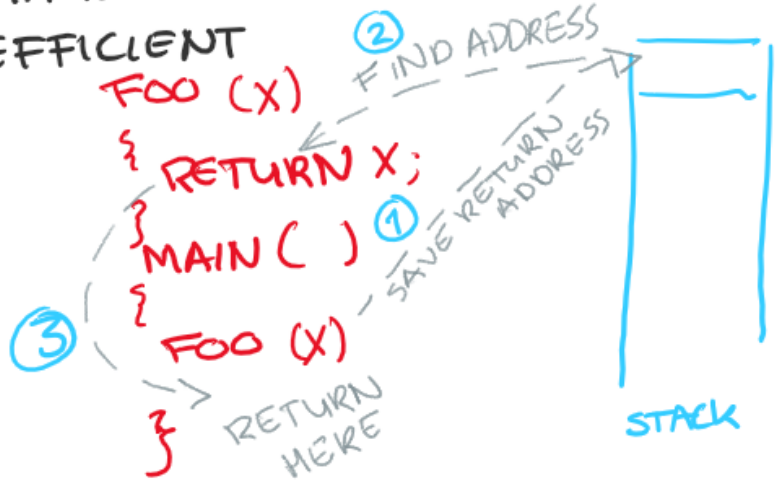
(ASSUMING YOU FOUND ALL THE APPROPRIATE PLACES)

(33) ANOTHER SOLUTION WAS SUGGESTED BY GOOGLE

RETPOLINE

A CLEVER IDEA TO MAKE BP WORK FOR YOU

IT IS RELATED TO HOW MODERN PROCESSORS MAKE RETURNING FROM A FUNCTION CALL EFFICIENT



THIS IS SO COMMON THAT TODAY'S CPUS HAVE OPTIMISED IT IN HARDWARE TO AID SPECULATIVE EXECUTION

— WHENEVER IT COMES ACROSS A FUNCTION CALL, SUCH AS FOO(X) IT SAVES THIS IN A SPECIAL CACHE

WHEN IT SPECULATIVELY ENCOUNTERS A RETURN IT WILL TAKE THE RETURN ADDRESS FROM THAT CACHE AND RESUME SPECULATIVE EXECUTION THERE

CLEVER.

SO WE TRANSLATE PROBLEMATIC INDIRECT JUMPS SUCH AS

```
JMP *EAX
```

TO:

```
    CALL   SETUP_TARGET
SPECTRE_TRAP:
    PAUSE
    JMP  SPECTRE_TRAP
SETUP_TARGET:
    MOV %RAX, (ESP)
    RET
```

} LOOP FOREVER IN SPECULATIVE BRANCH

→ MOV TARGET ADDR TO TOP OF STACK

WHAT WILL THIS DO?

```
CALL   SETUP_TARGET
SPECTRE_TRAP:
    PAUSE
    JMP SPECTRE_TRAP

SETUP_TARGET:
    MOV %RAX, (ESP)
    RET
```

① PUTS ADDR OF NEXT INSTRUCTION IN THIS SPECIAL "CACHE"

② NOW CHANGE THE RETURN ADDRESS ON THE STACK WITHOUT CHANGING IT IN THE SPECIAL CACHE

③ SPECULATION CONTINUES HERE (INFINITE LOOP)

UNTIL THE REAL TARGET IS FOUND

→ SPECULATION IS SQUASHED

→ CONTINUE AT *RAX

WE ARE DONE

→ WE CAN WRITE A COMPILER PASS TO MODIFY THE INDIRECT JUMPS AND CALLS TO THE ONES USING RETPOLINES

UNFORTUNATELY, NONE OF THE SOLUTIONS SOLVES ALL PROBLEMS ON ALL PROCESORS

AND NONE OF THEM IS FREE!

EXAMPLE 2

Bug-free Exploitation in Browsers

Dedup Est Machina

Published at IEEE S&P 2016

with Erik, Kaveh, Cristiano

Won **Pwnie Award** at Black HAT 2016

black hat USA 2016

"Most Innovative Research"

Exploit of Microsoft Edge browser on Windows 10 from malicious JavaScript

...without relying on a single software bug

Dedup Est Machina

Erik Bosman

Kaveh Razavi

Herbert Bos

Cristiano Giuffrida

Dedup Est Machina

Memory deduplication
(software side channel)

Dedup Est Machina

Memory deduplication
(software side channel)
+
Rowhammer
(hardware glitch)

Dedup Est Machina

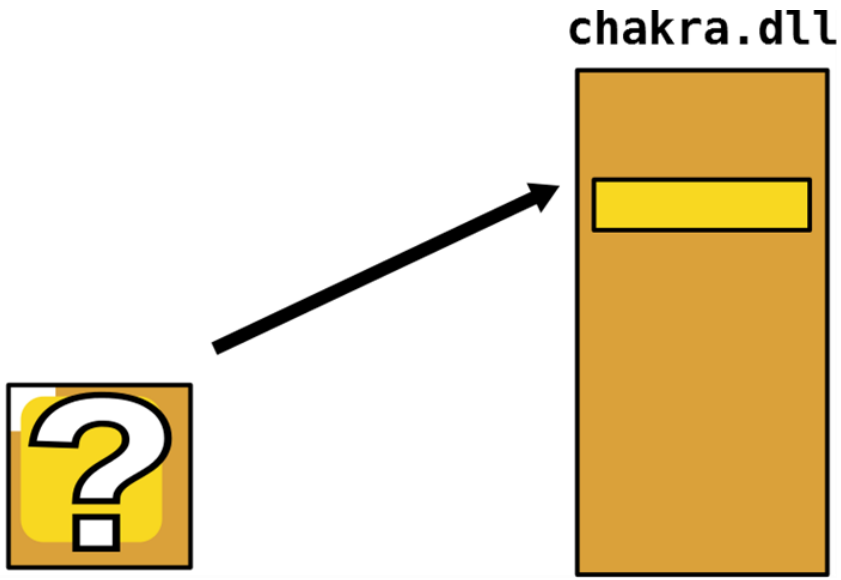
Memory deduplication
(software side channel)
+
Rowhammer
(hardware glitch)

Exploit MS Edge without software bugs
(from JavaScript)

Dedup Est Machina: Overview

Memory deduplication

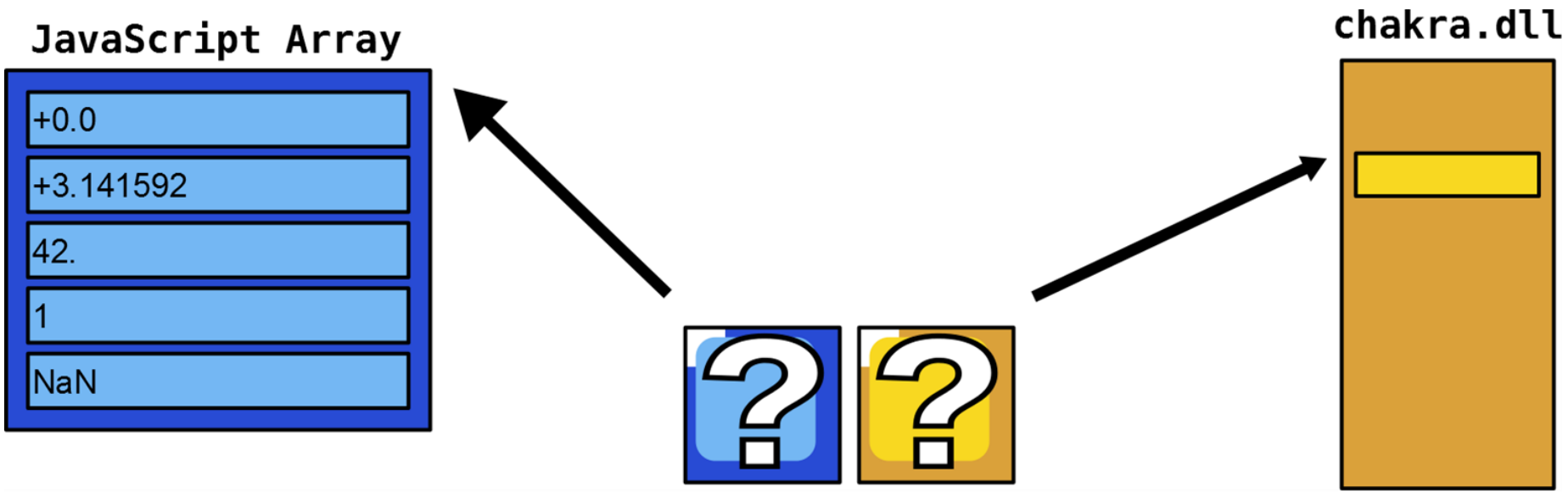
Leak randomized heap and code pointers



Dedup Est Machina: Overview

Memory deduplication

Leak randomized heap and code pointers

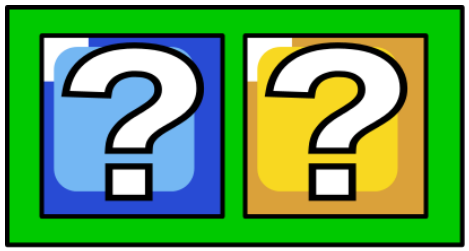


Dedup Est Machina: Overview

Memory deduplication

Leak randomized heap and code pointers

Create a fake JavaScript object



Dedup Est Machina: Overview

Memory deduplication

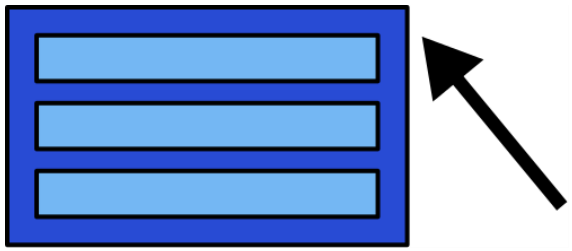
Leak randomized heap and code pointers

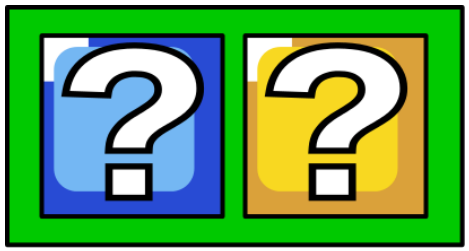
Create a fake JavaScript object

+

Rowhammer

Create a reference to our fake object





Dedup Est Machina: Overview

Memory deduplication

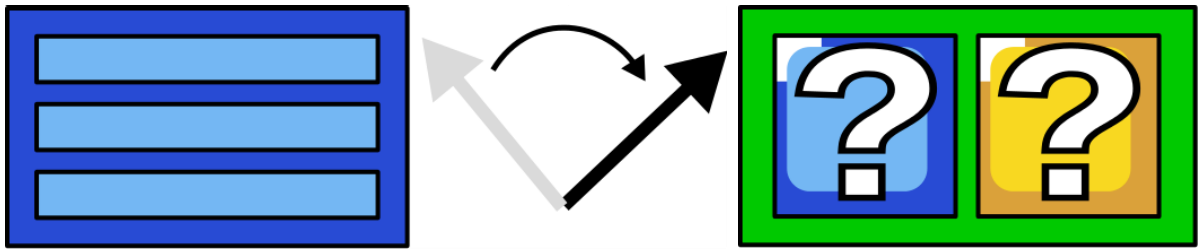
Leak randomized heap and code pointers

Create a fake JavaScript object

+

Rowhammer

Create a reference to our fake object



Dedup Est Machina: Overview

Memory deduplication

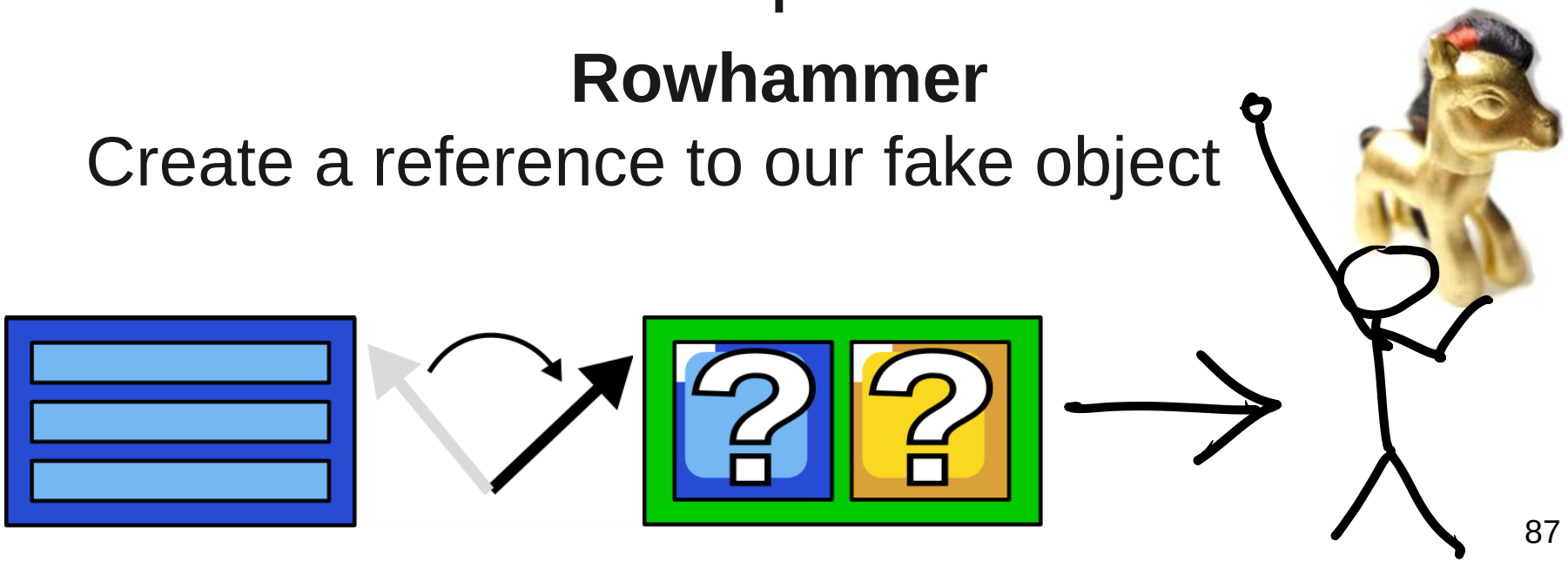
Leak randomized heap and code pointers

Create a fake JavaScript object

+

Rowhammer

Create a reference to our fake object



Memory Deduplication

A strategy to reduce physical memory usage

Removes duplication in physical memory

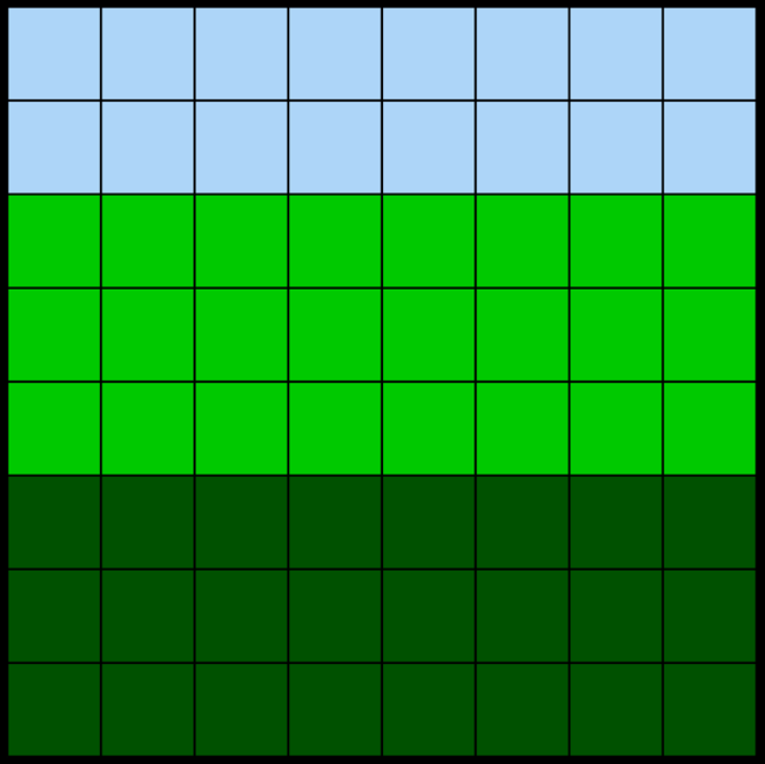
Common in virtualization environments

Enabled by **default on Windows**

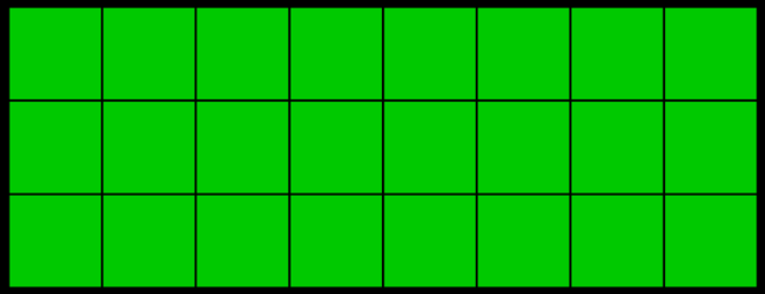
- Windows 8.1
- Windows 10

Memory Deduplication: Mechanics

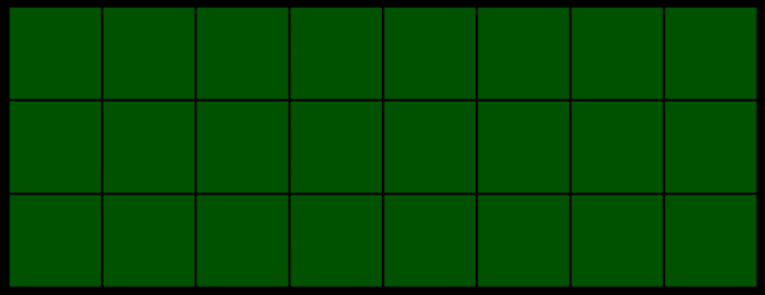
physical memory



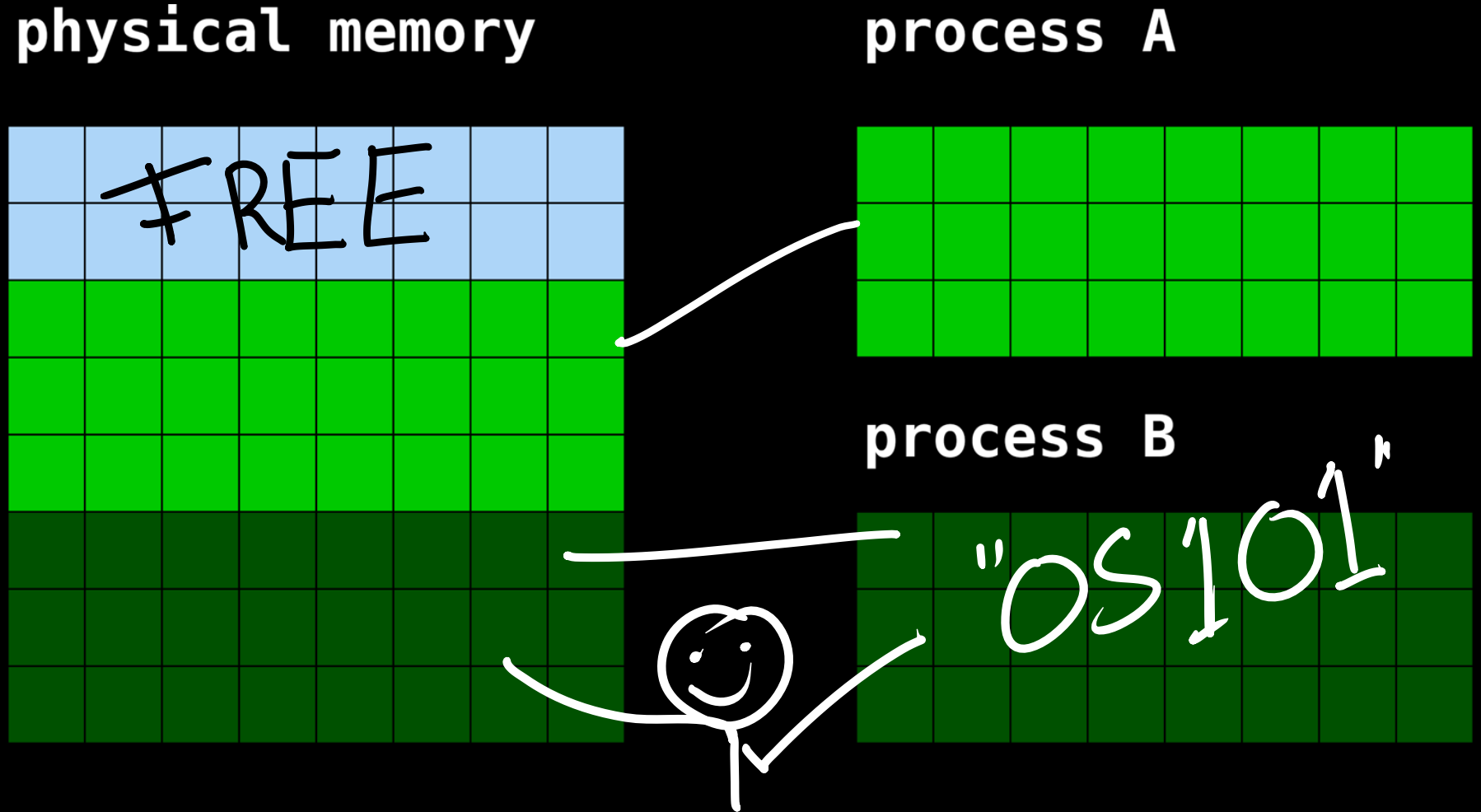
process A



process B

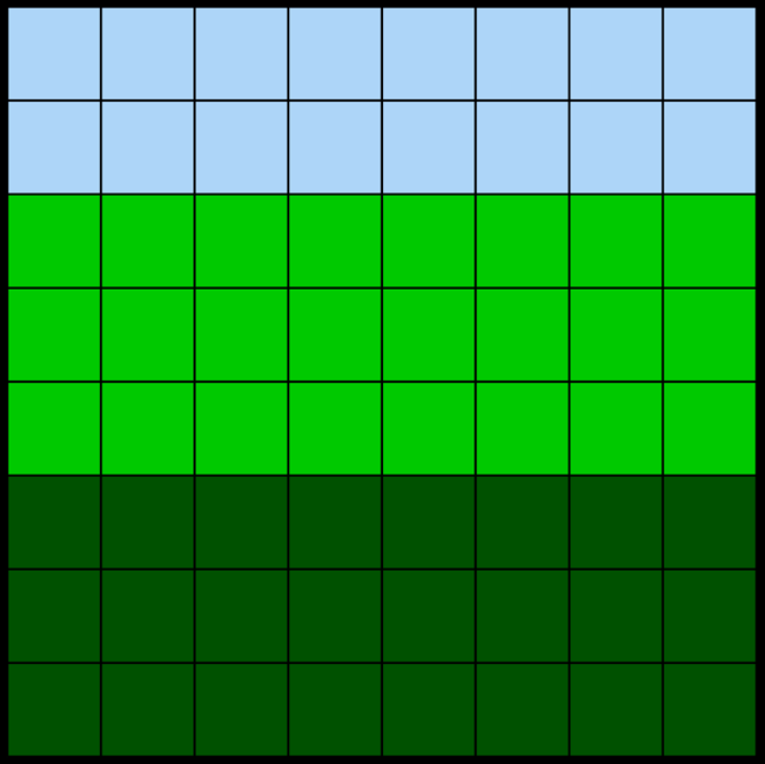


Memory Deduplication: Mechanics

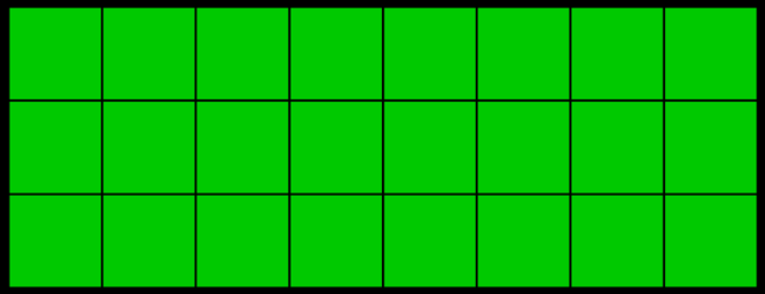


Memory Deduplication: Mechanics

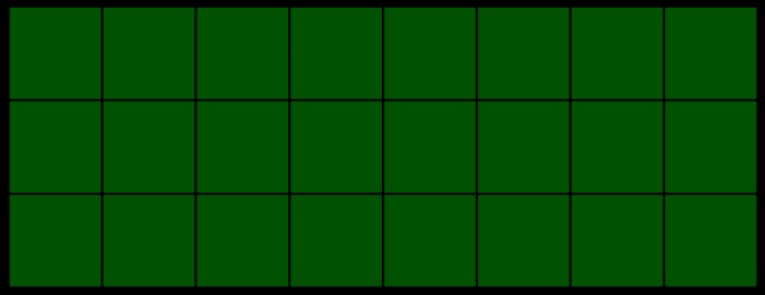
physical memory



process A

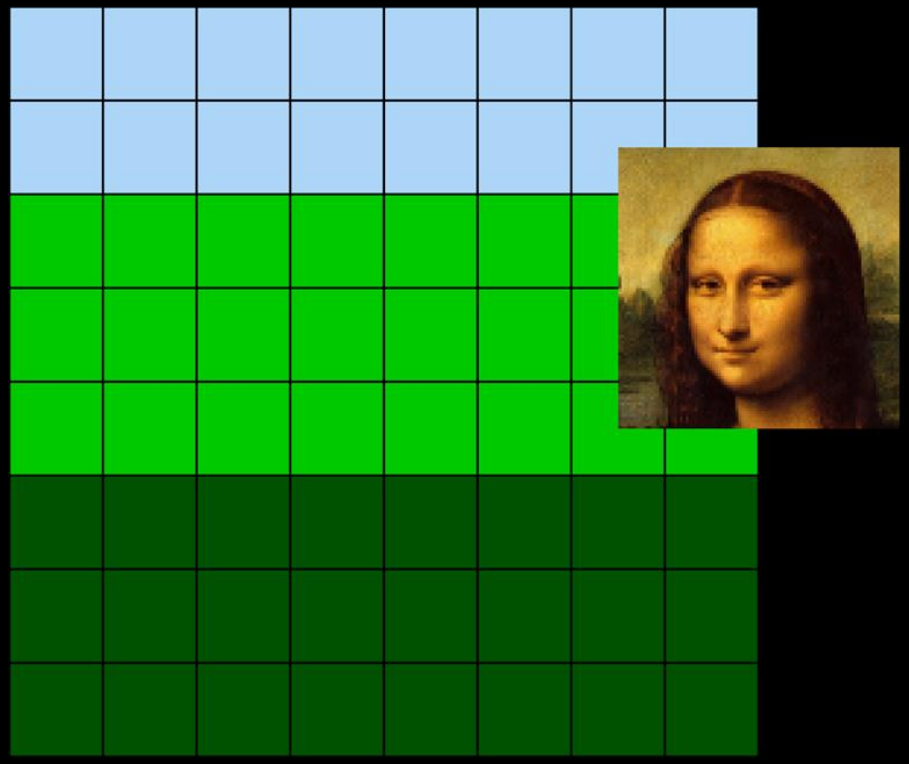


process B

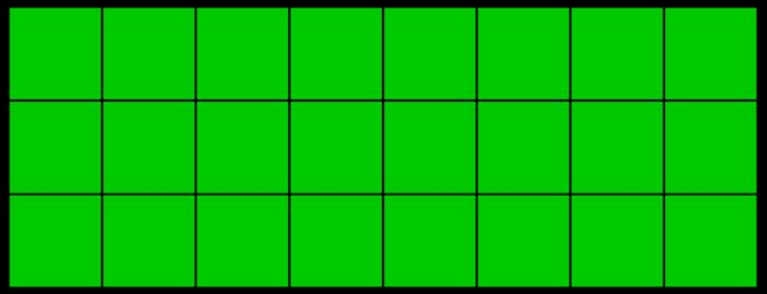


Memory Deduplication: Mechanics

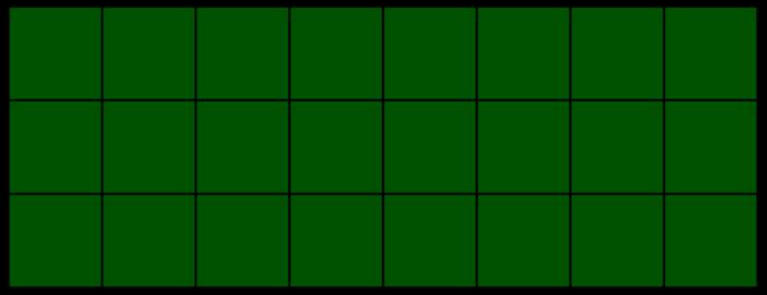
physical memory



process A

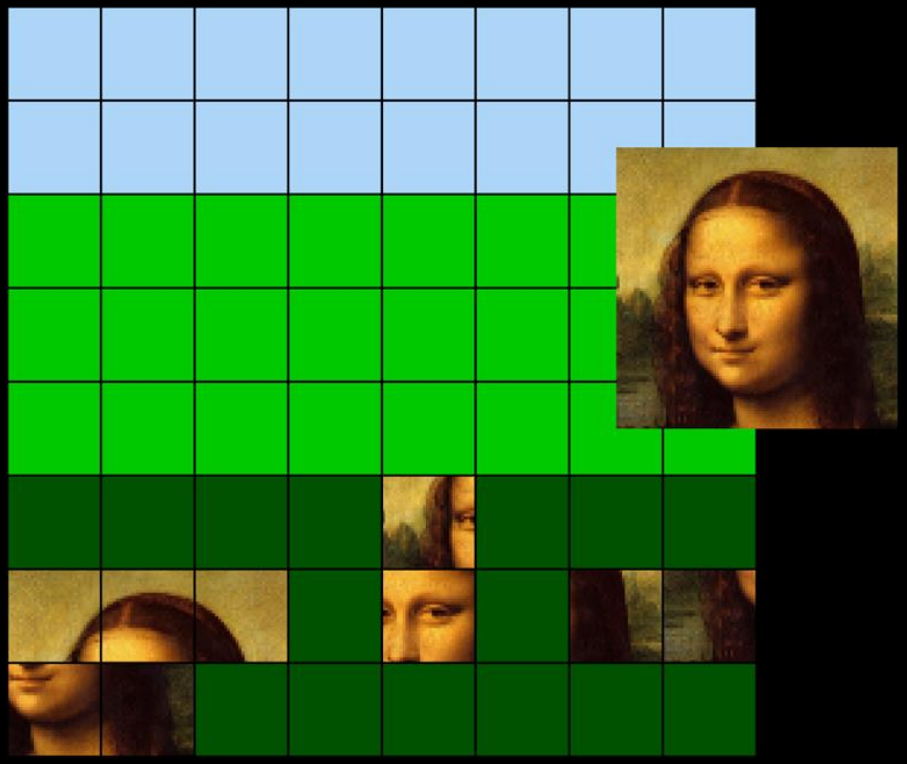


process B

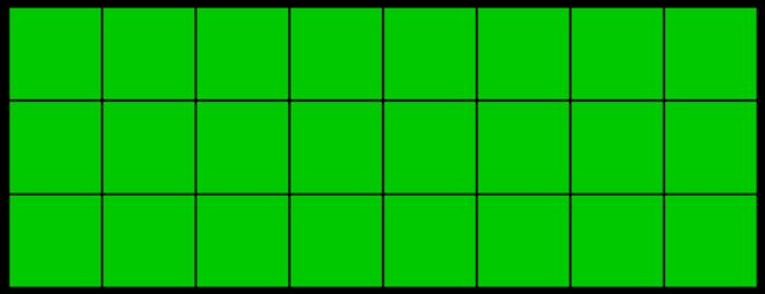


Memory Deduplication: Mechanics

physical memory



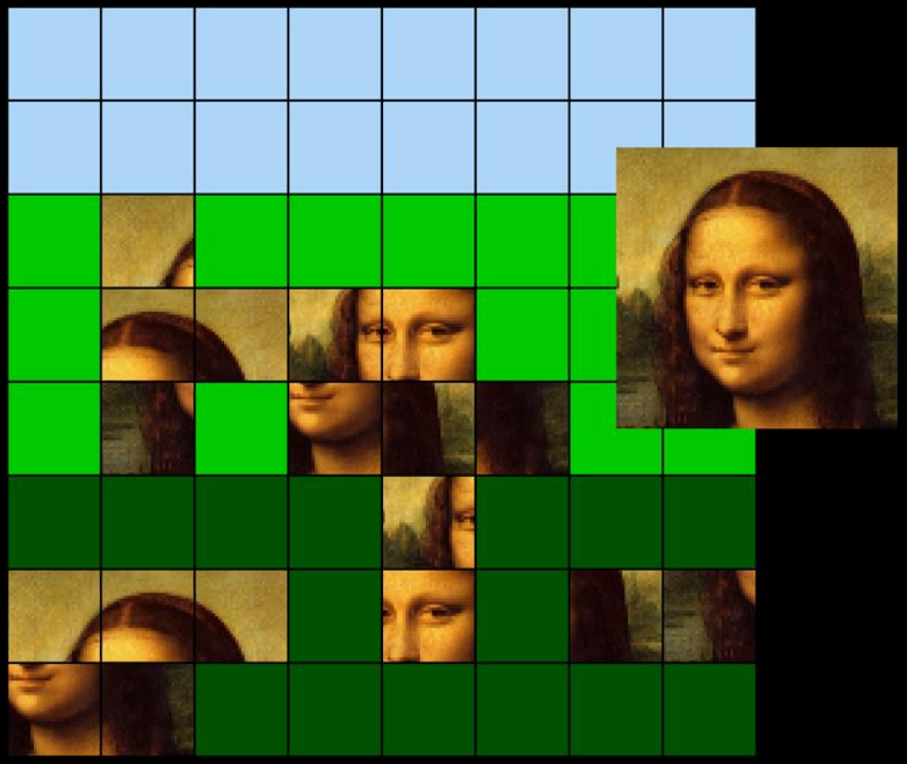
process A



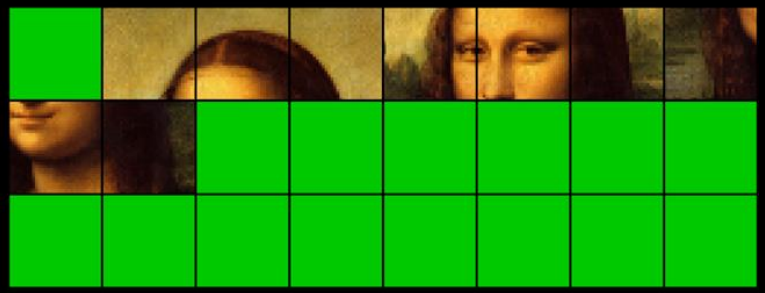
process B

Memory Deduplication: Mechanics

physical memory



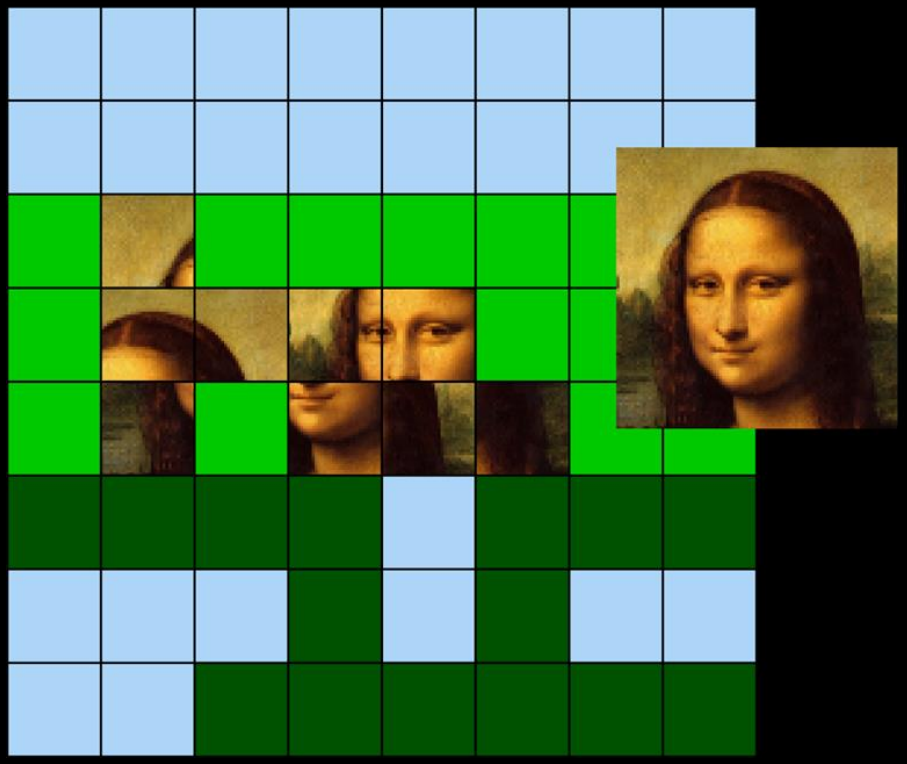
process A



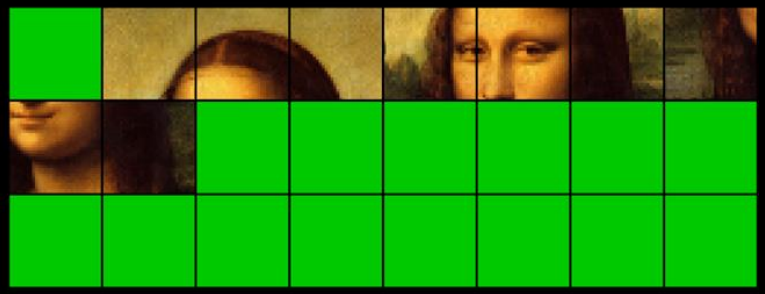
process B

Memory Deduplication: Mechanics

physical memory



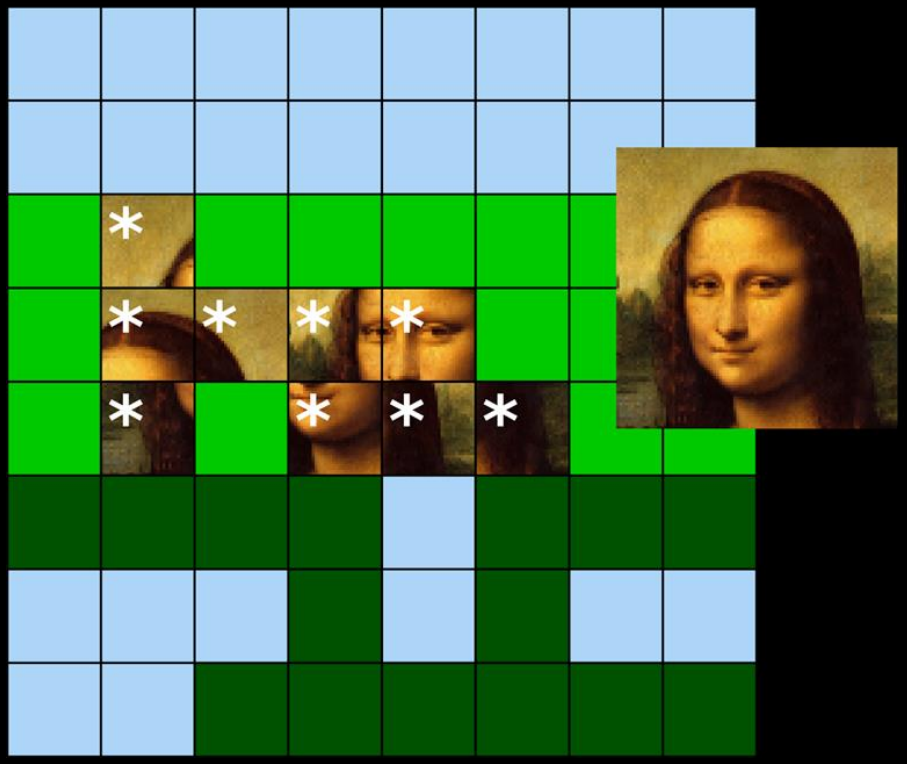
process A



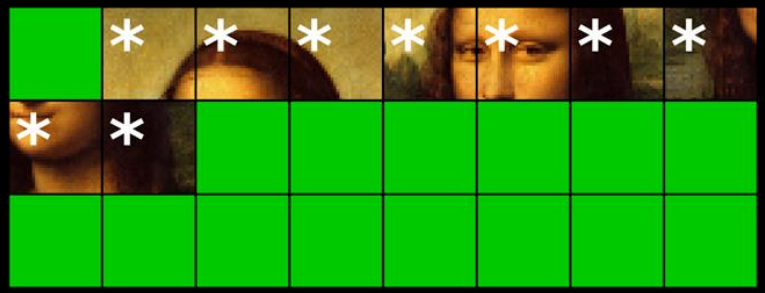
process B

Memory Deduplication: Mechanics

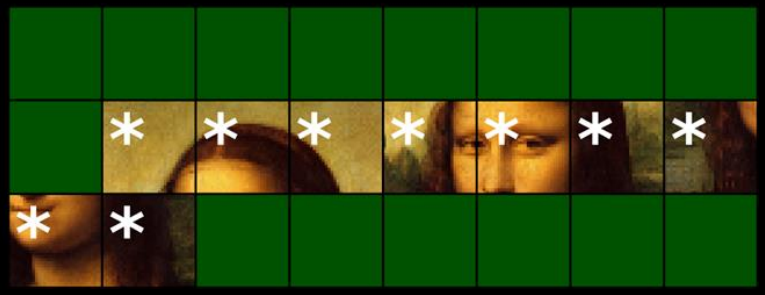
physical memory



process A

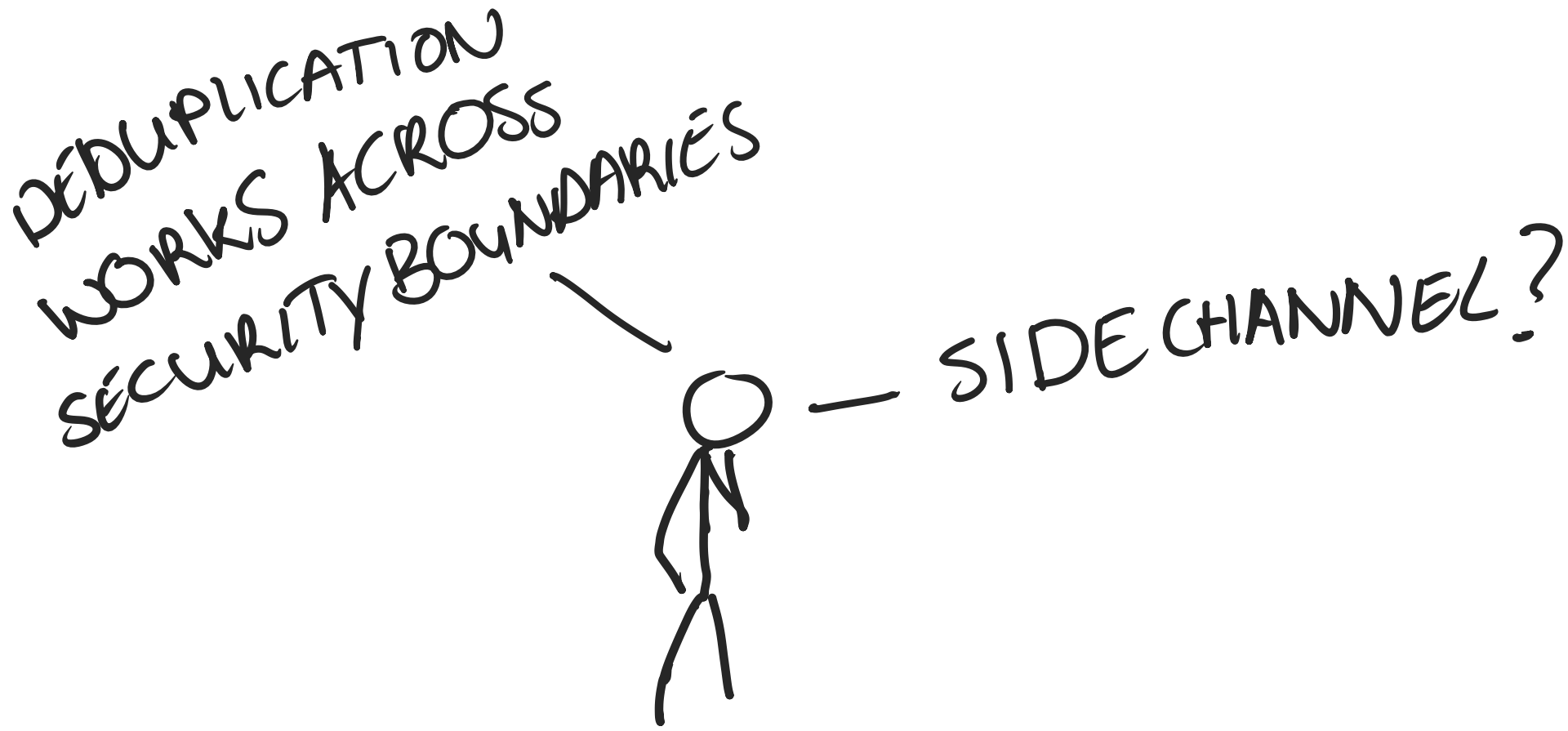


process B



Memory Deduplication: The Problem

Memory Deduplication: The Problem



Memory Deduplication: Timing Side Channel

normal write

Memory Deduplication: Timing Side Channel

normal write

Memory Deduplication: Timing Side Channel

normal write

copy on write (due to deduplication)

Memory Deduplication: Timing Side Channel

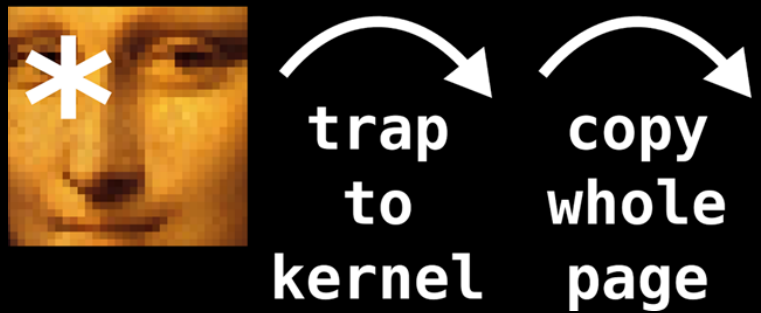
normal write

copy on write (due to deduplication)

Memory Deduplication: Timing Side Channel

normal write

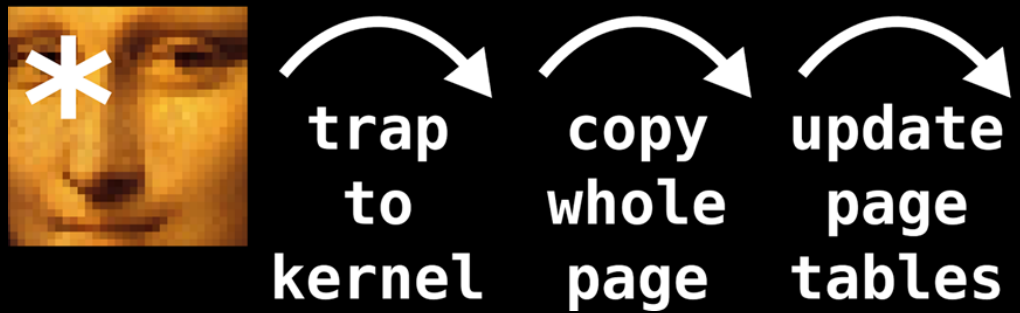
copy on write (due to deduplication)



Memory Deduplication: Timing Side Channel

normal write

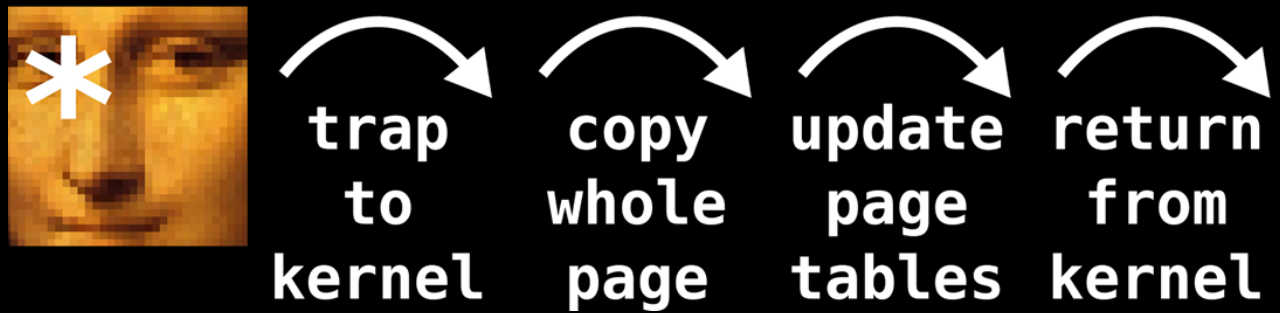
copy on write (due to deduplication)



Memory Deduplication: Timing Side Channel

normal write

copy on write (due to deduplication)

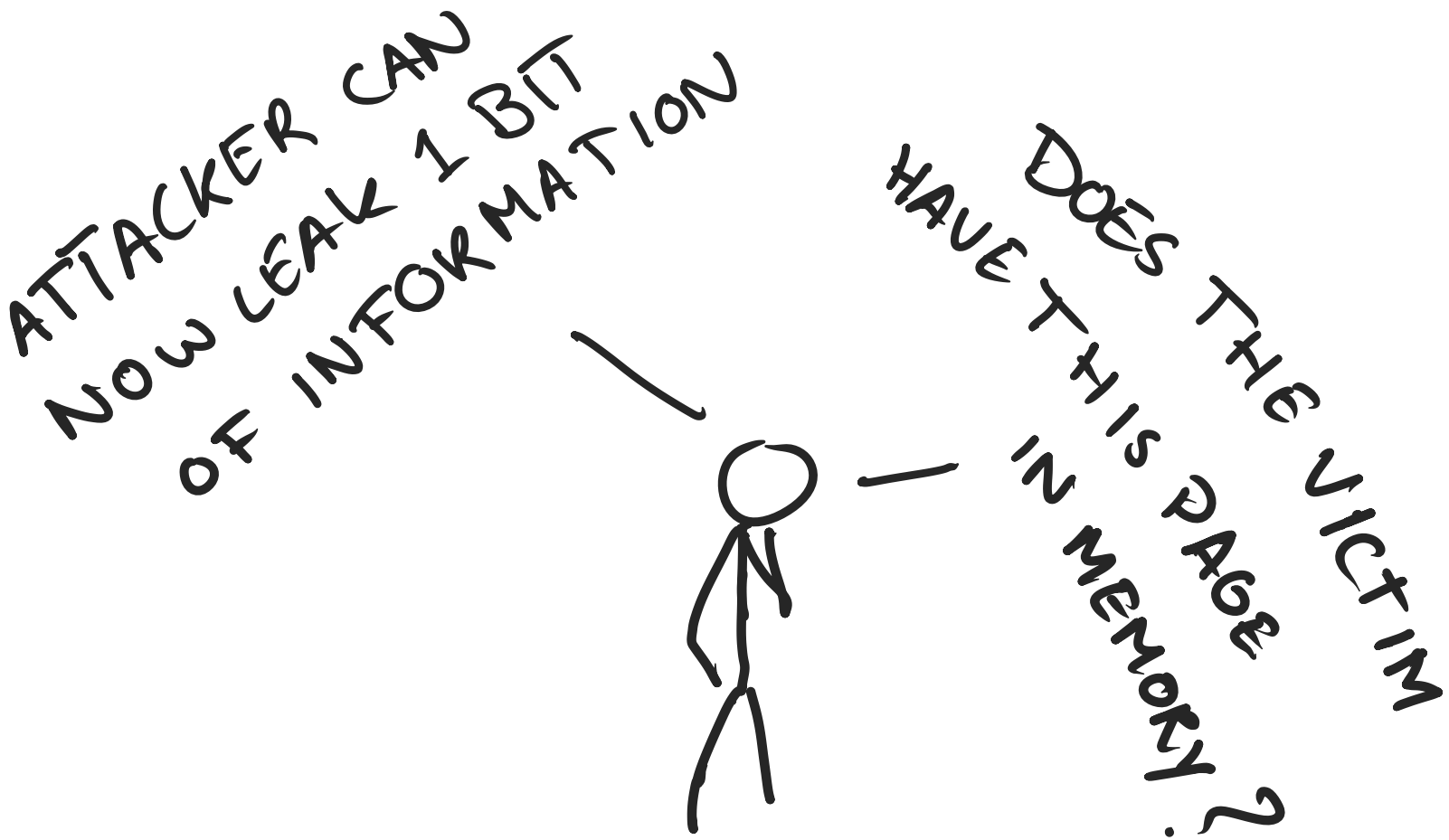


Memory Deduplication: Timing Side Channel

normal write

copy on write (due to deduplication)

Memory Deduplication: The Problem



Memory Deduplication: Side-channel Leaks

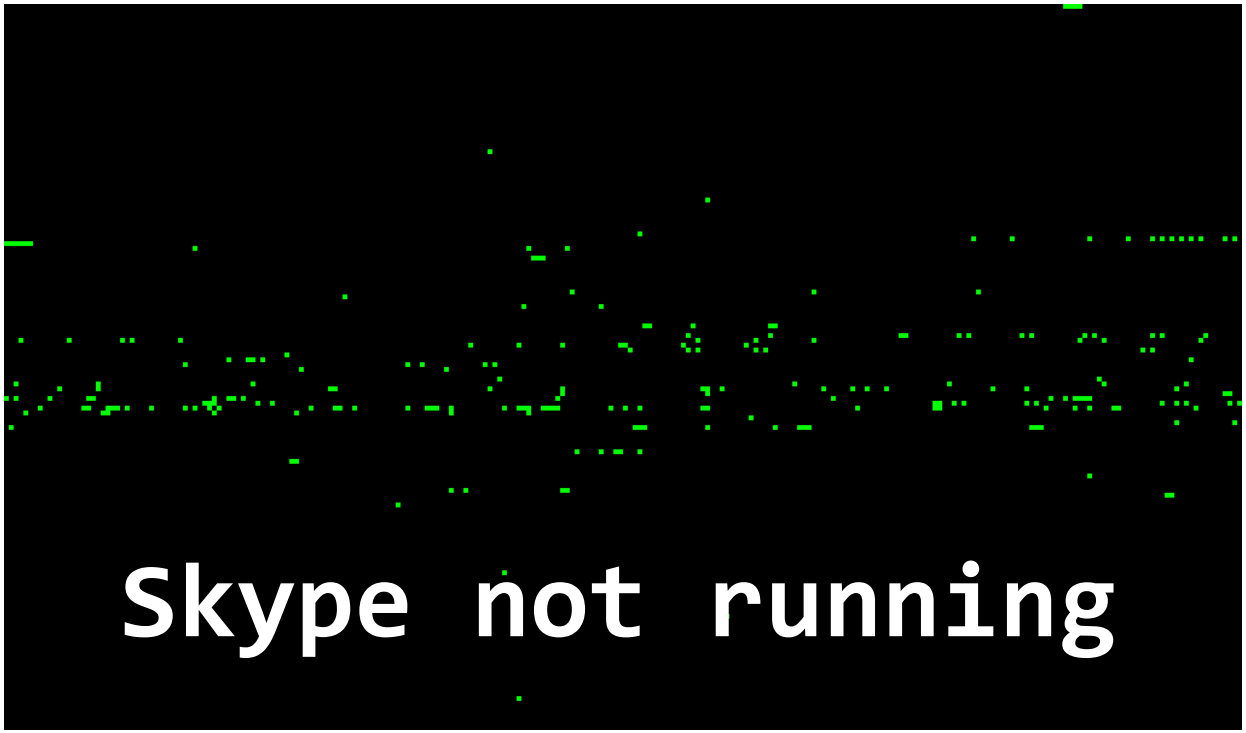
Very **coarse-grained**. Still interesting?

Is user logged into bank website X?

Memory Deduplication: Side-channel Leaks

Very **coarse-grained**. Still interesting?

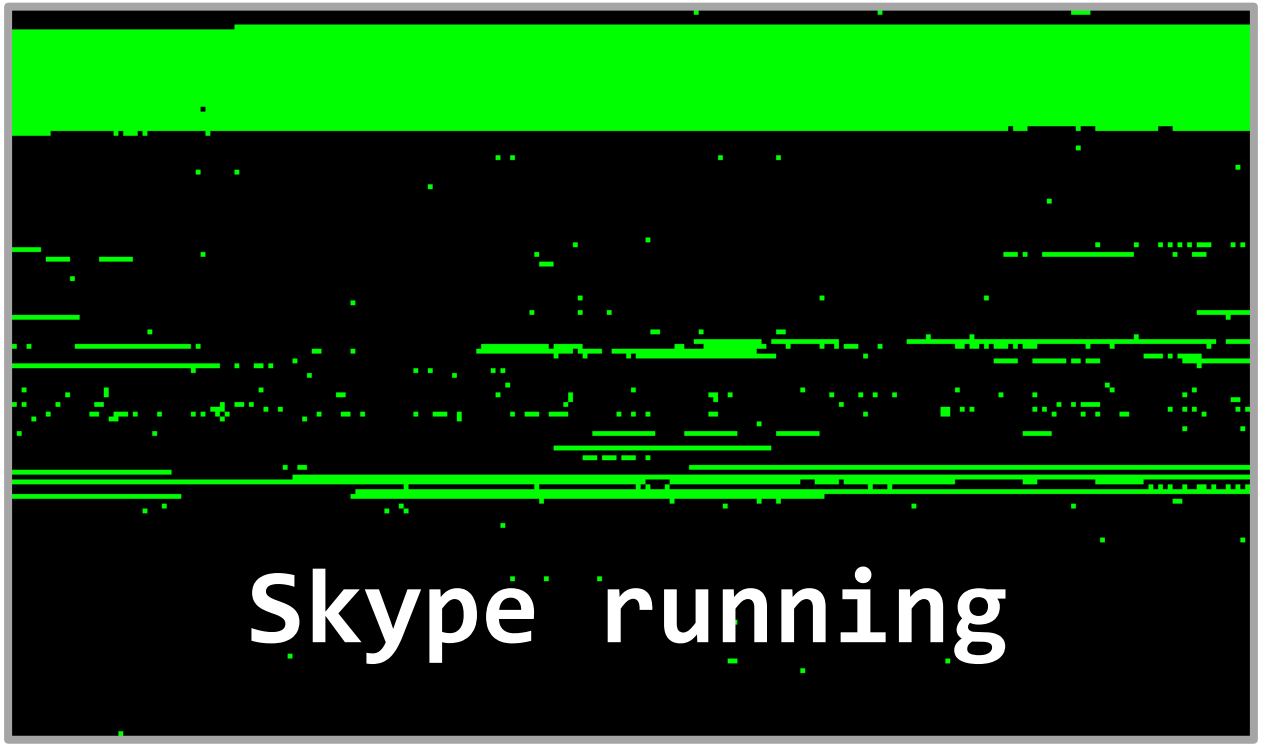
Is user running software X?



Memory Deduplication: Side-channel Leaks

Very **coarse-grained**. Still interesting?

Is user running software X?



Memory Deduplication: Software Exploitation

For software exploitation, 1 bit won't really cut it (e.g., need to leak 64-bit pointers for MS Edge)

*"Can we generalize this to leaking **arbitrary data** like randomized pointers or passwords?"*

Dedup Est Machina: Challenges

Challenge 1:

The secret we want to leak does not span an entire memory page

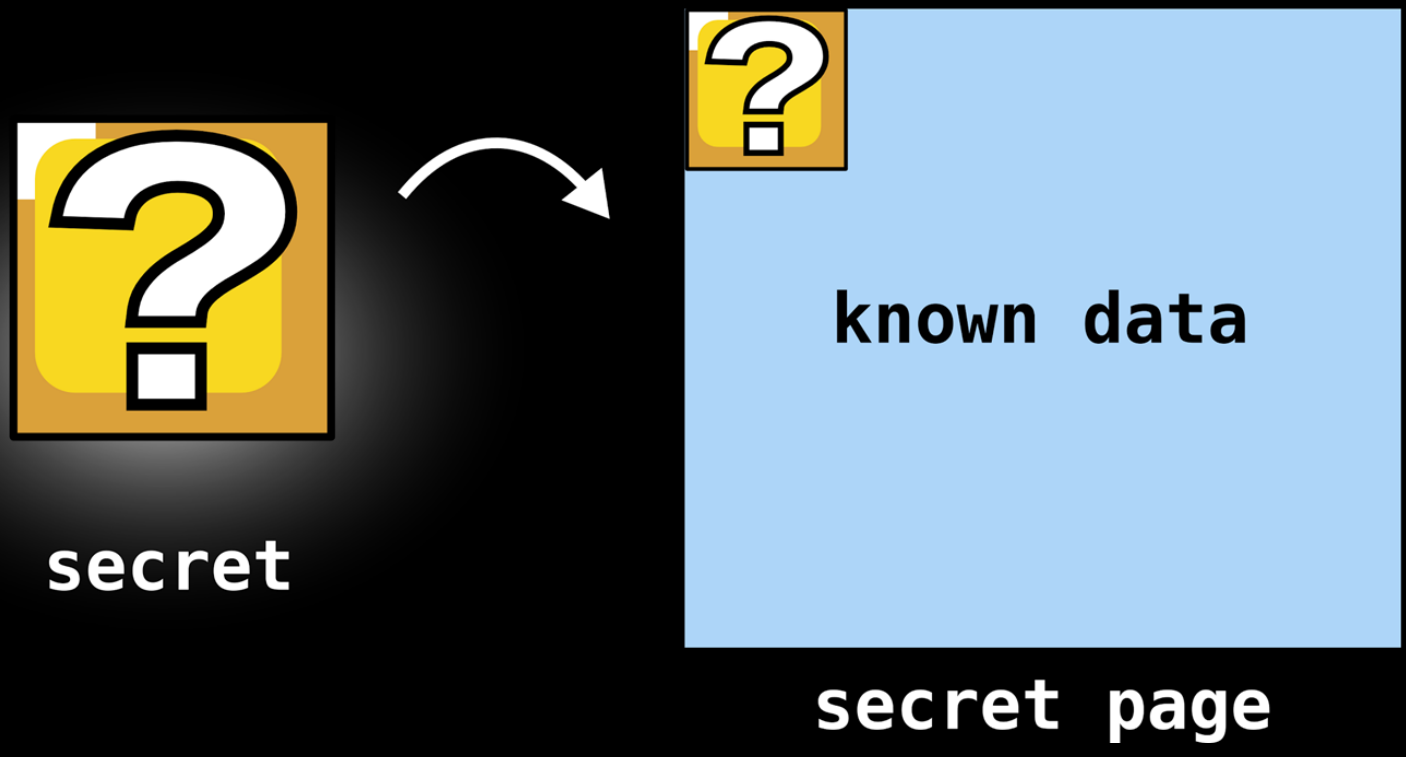
Dedup Est Machina: Challenges

Turning a secret into a page

secret

Dedup Est Machina: Challenges

Turning a secret into a page



Dedup Est Machina: Challenges

Challenge 2:

The secret to leak has too much entropy to leak it all at once

Dedup Est Machina: Challenges

Challenge 2:

The secret to leak has too much entropy to leak it all at once

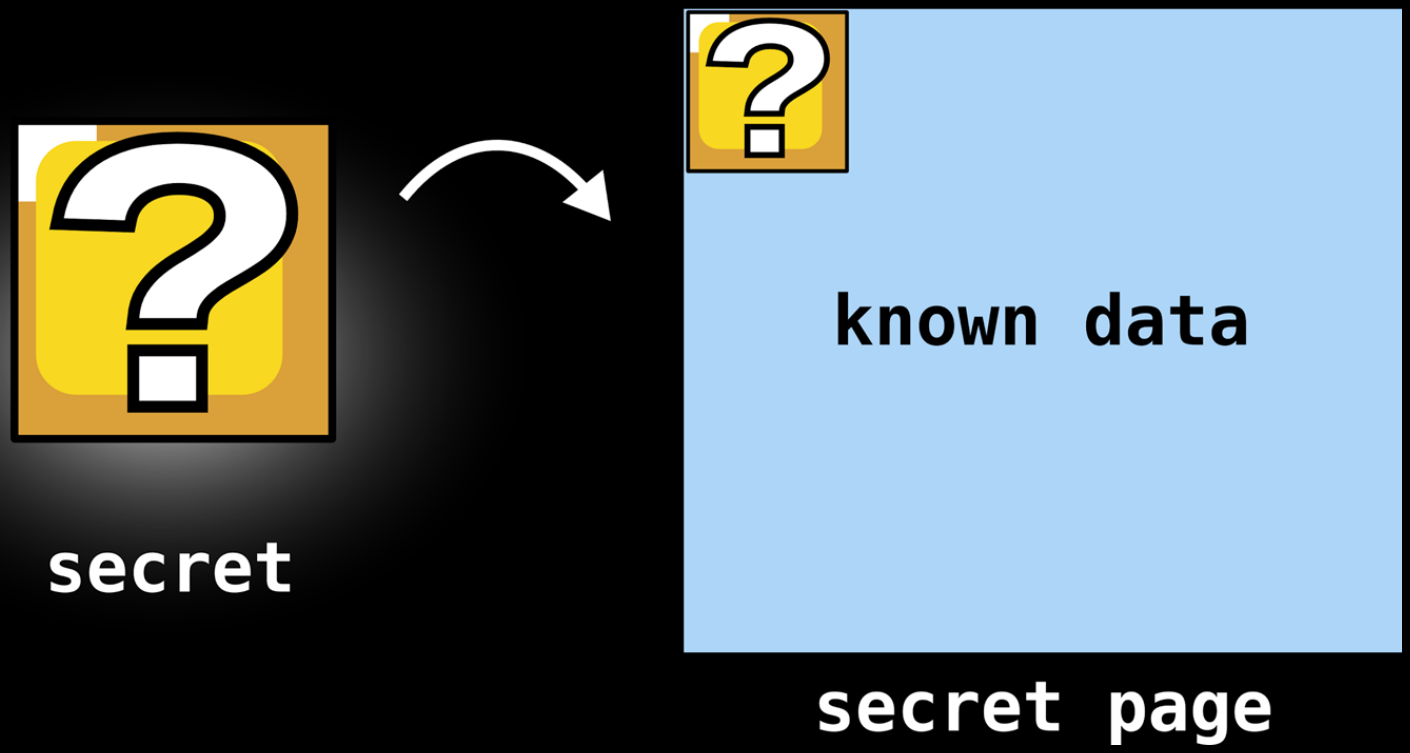
Primitive #1

Primitive #2

Primitive #3

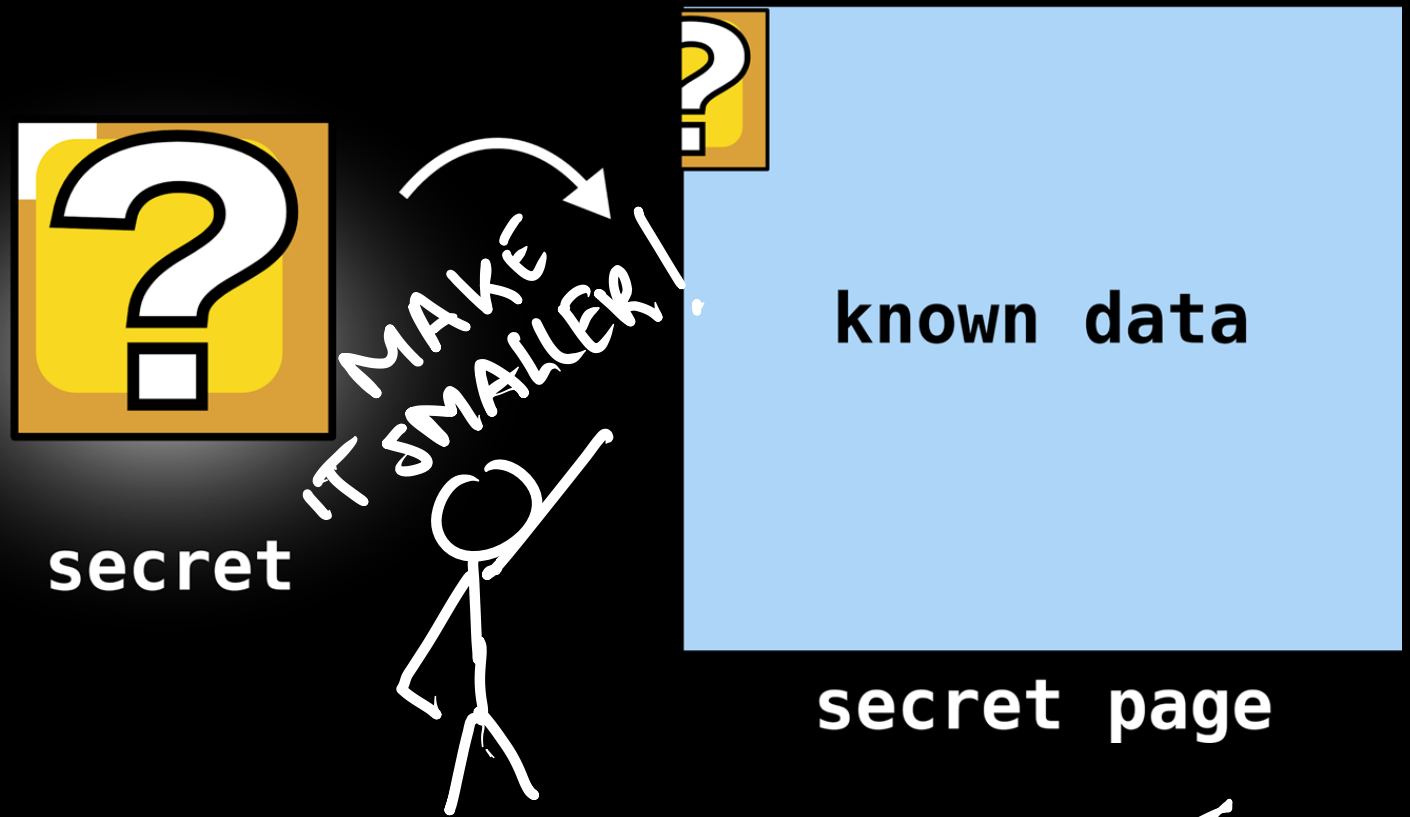
Dedup Est Machina: Primitives

Primitive #1: Alignment Probing



Dedup Est Machina: Primitives

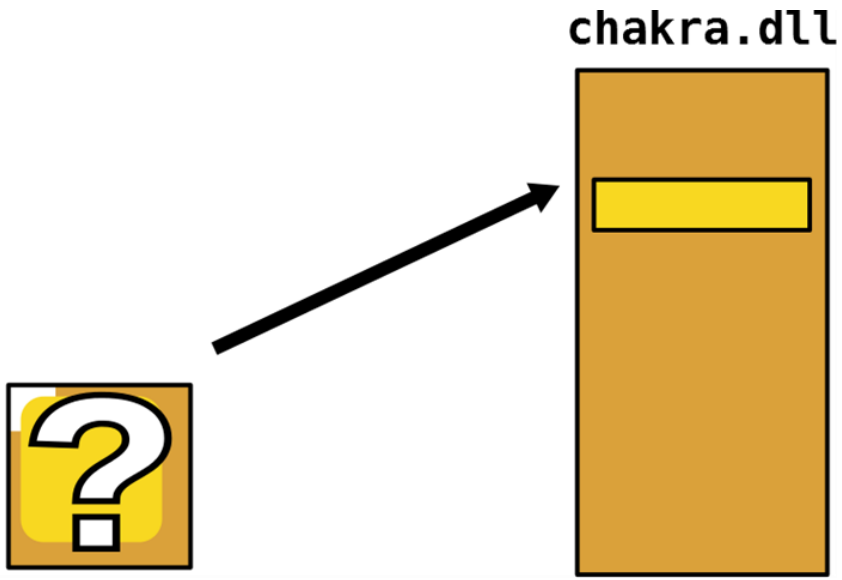
Primitive #1: Alignment Probing



Dedup Est Machina: Overview

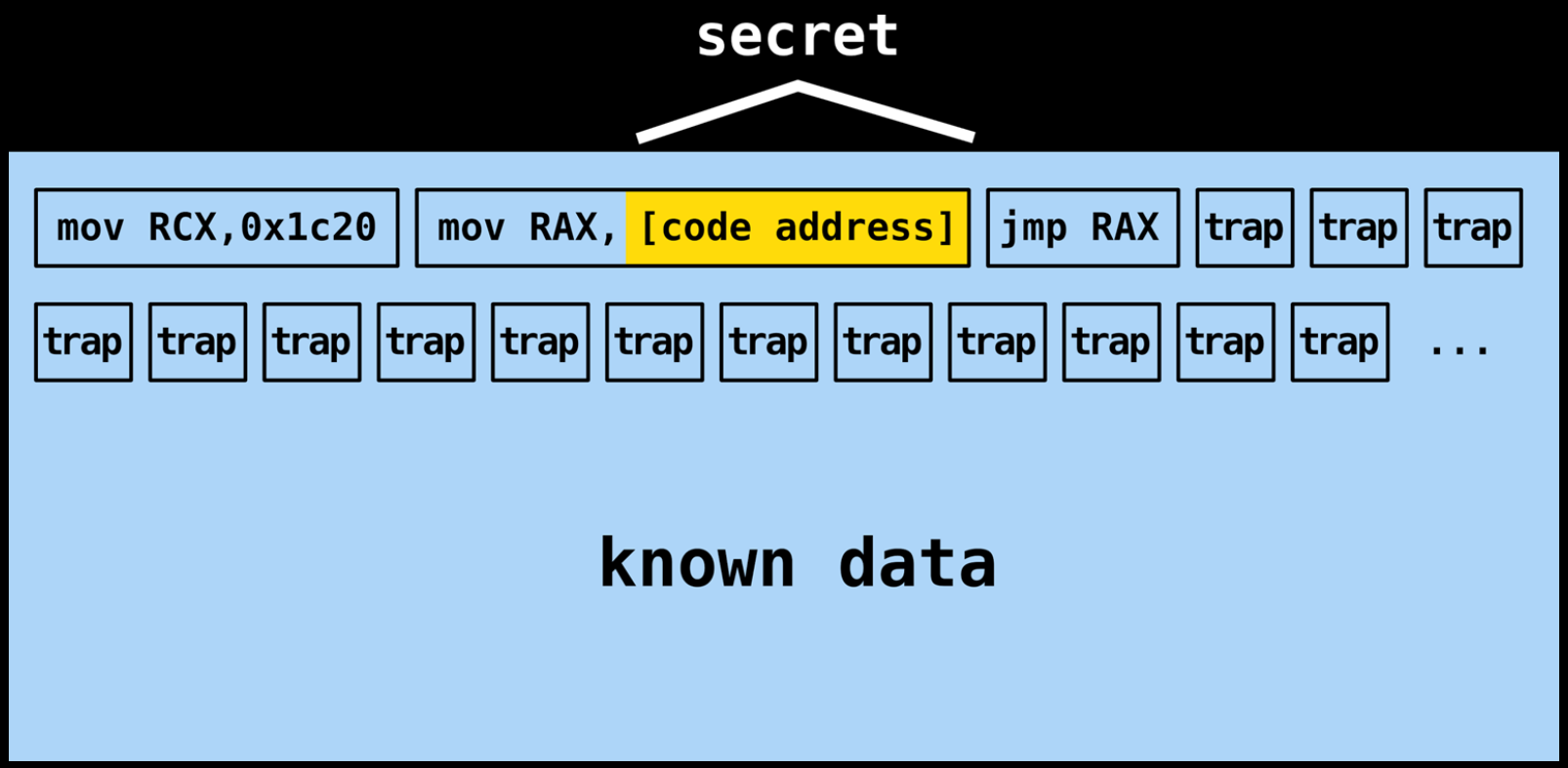
Memory deduplication

Leak randomized heap and code pointers



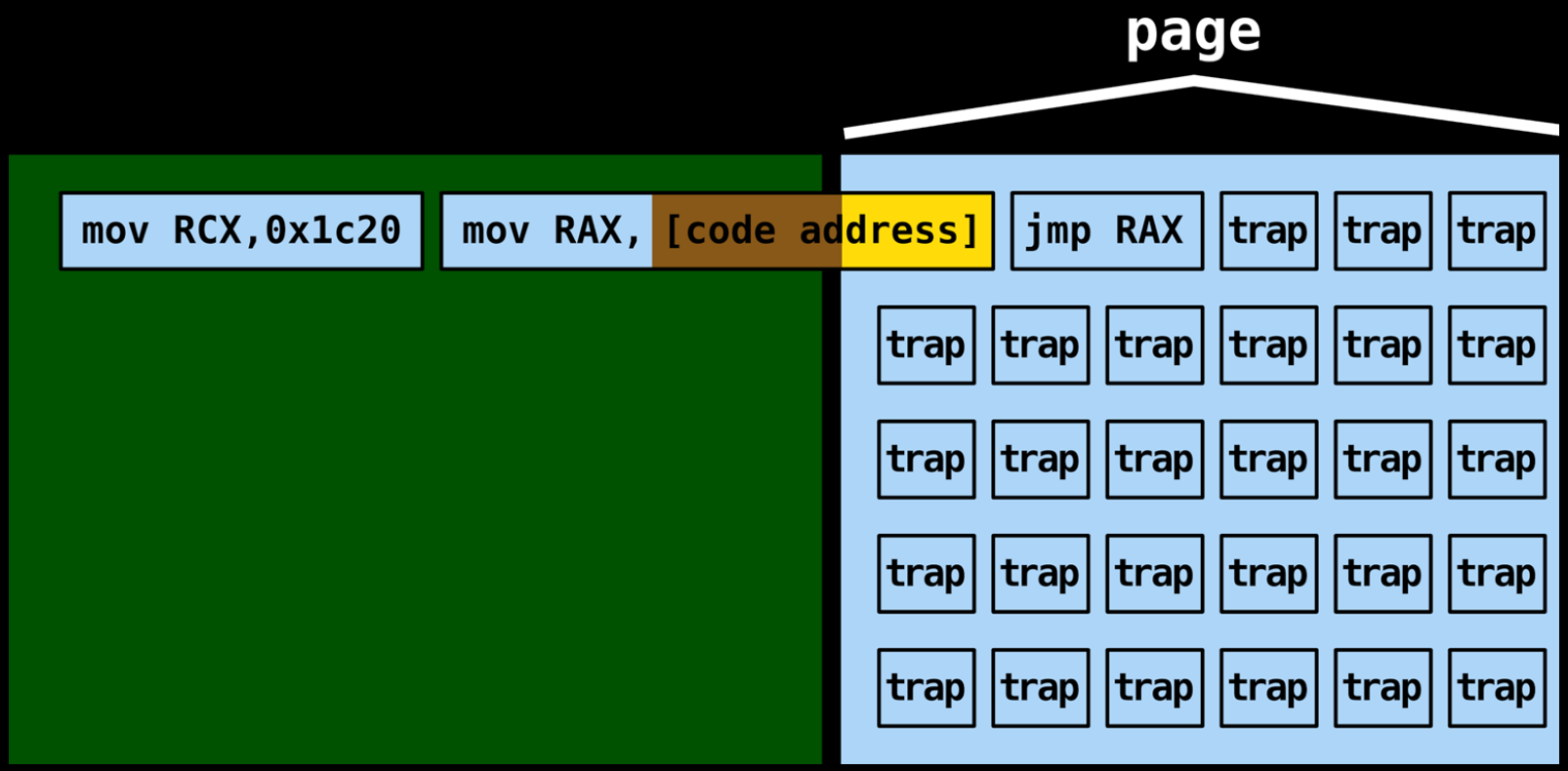
Dedup Est Machina: Leaking Code Pointer (#1)

JIT Function Epilogue in MS Edge



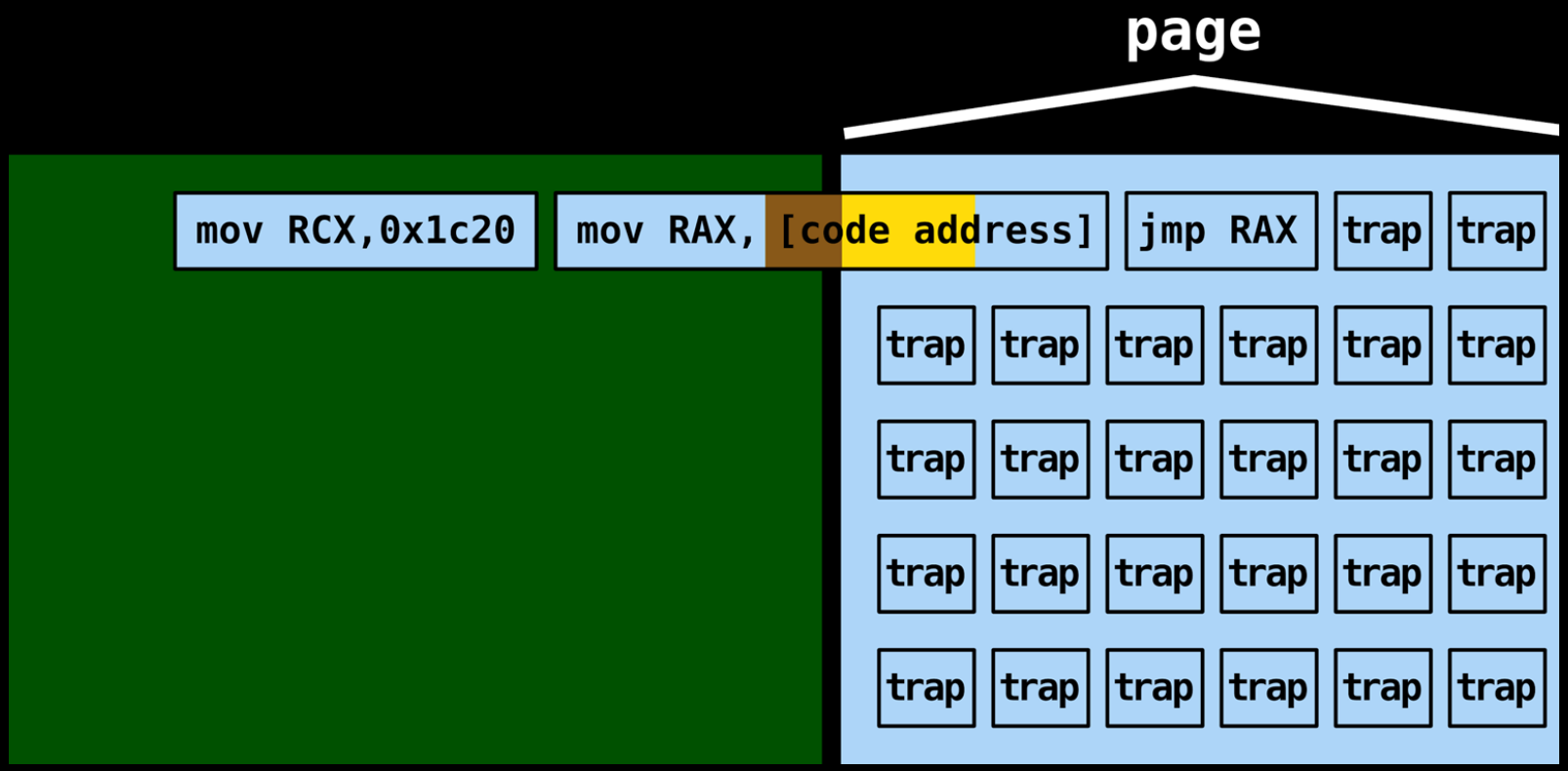
Dedup Est Machina: Leaking Code Pointer (#1)

JIT Function Epilogue in MS Edge



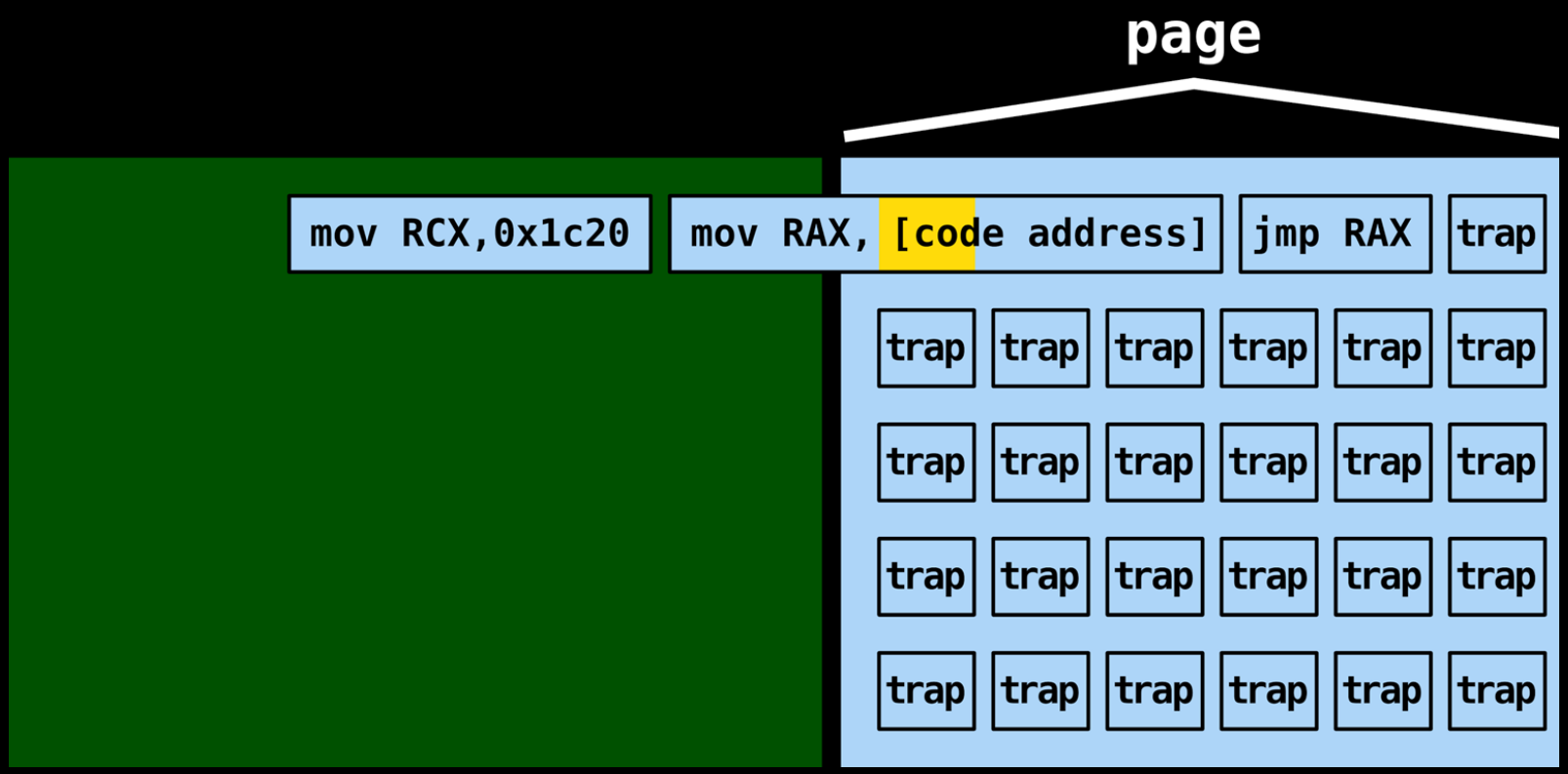
Dedup Est Machina: Leaking Code Pointer (#1)

JIT Function Epilogue in MS Edge



Dedup Est Machina: Leaking Code Pointer (#1)

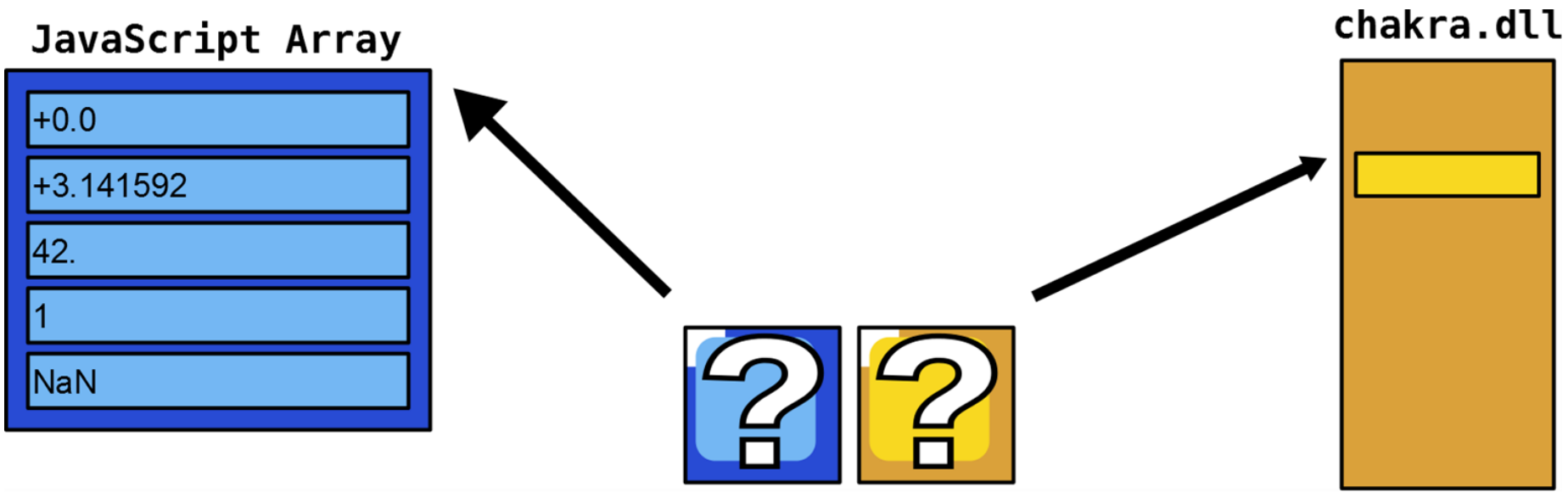
JIT Function Epilogue in MS Edge



Dedup Est Machina: Overview

Memory deduplication

Leak randomized heap and code pointers



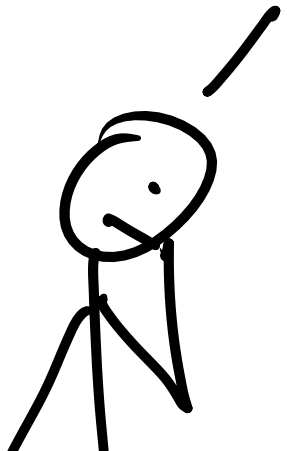
Dedup Est Machina: Leaking Heap Pointer

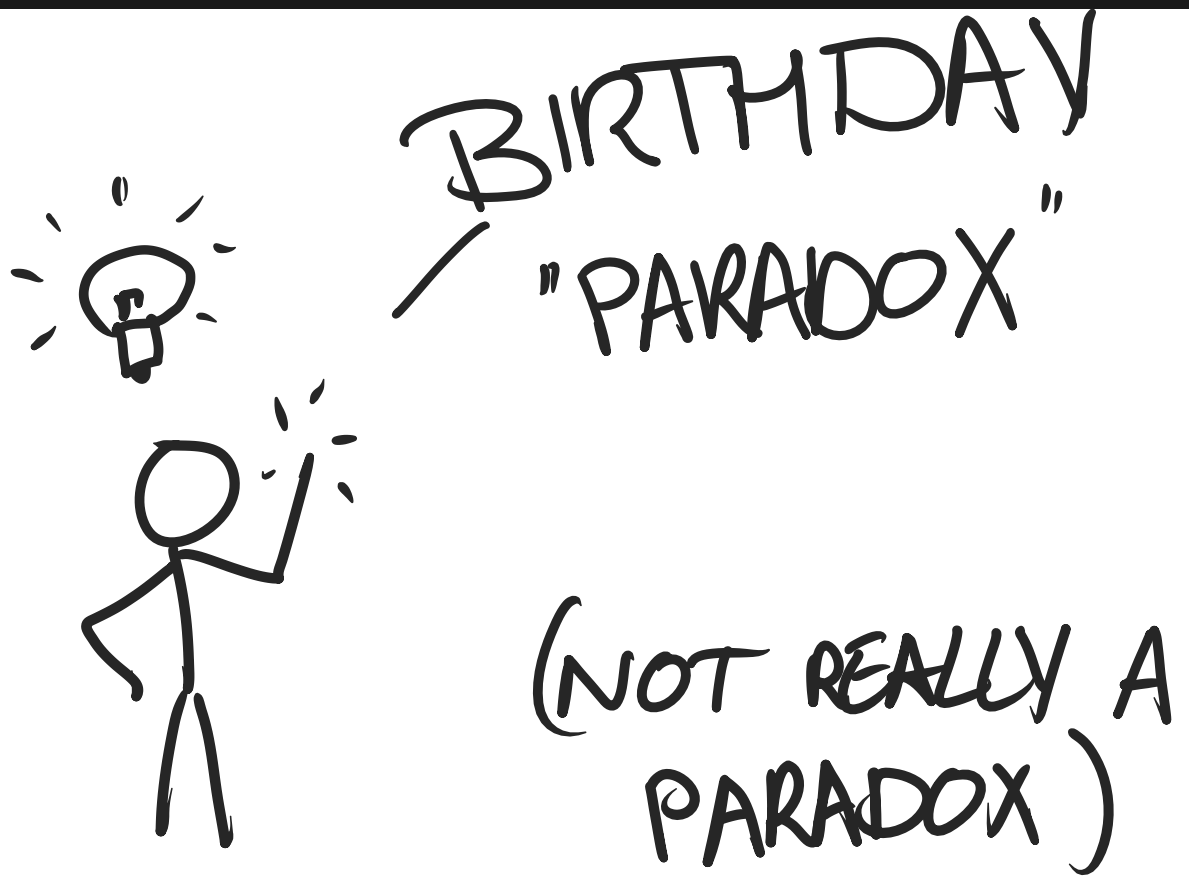
Heap pointers are word aligned

Alignment probing won't cut it, same for primitive #2

Time for primitive #3!

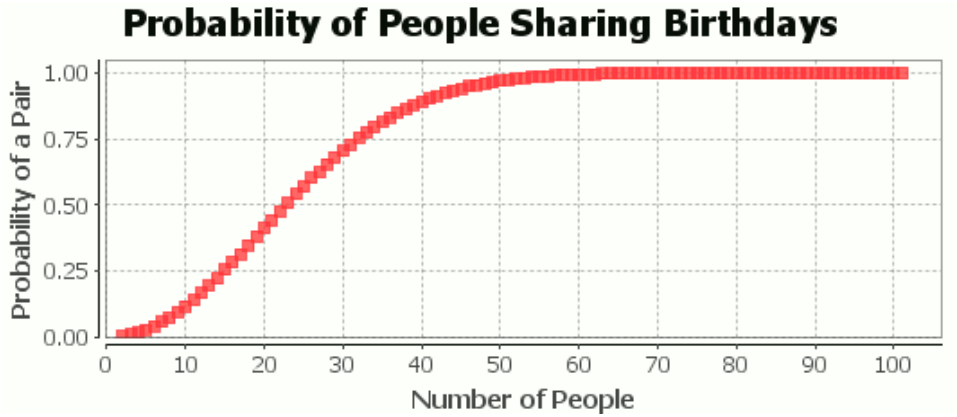
*"How do we leak a heap pointer if we can only leak the secret **all at once**?"*




BIRTHDAY "PARADOX"
(NOT REALLY A PARADOX)

Dedup Est Machina: Birthday Paradox

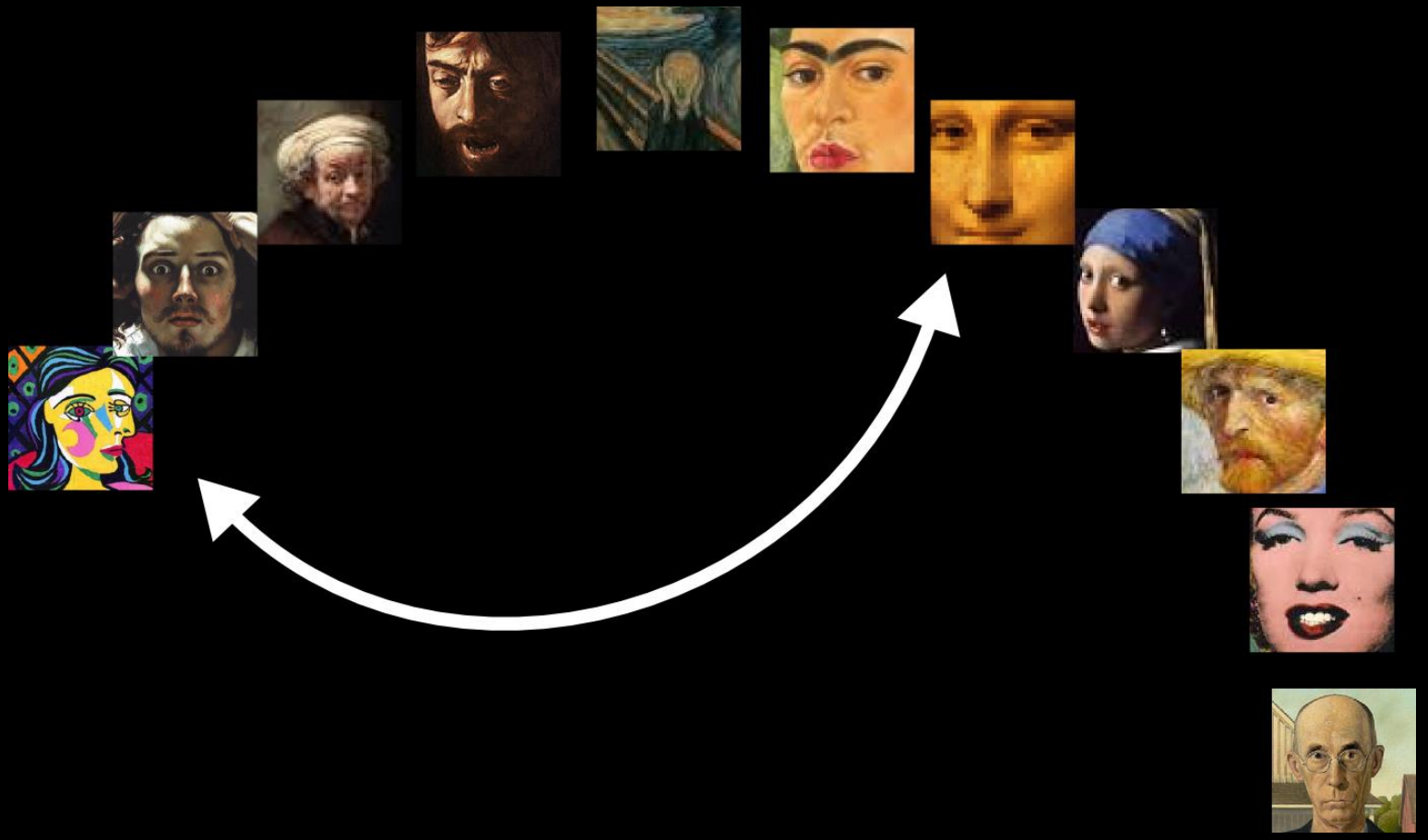
Only 23 people for a 50% same-birthday chance



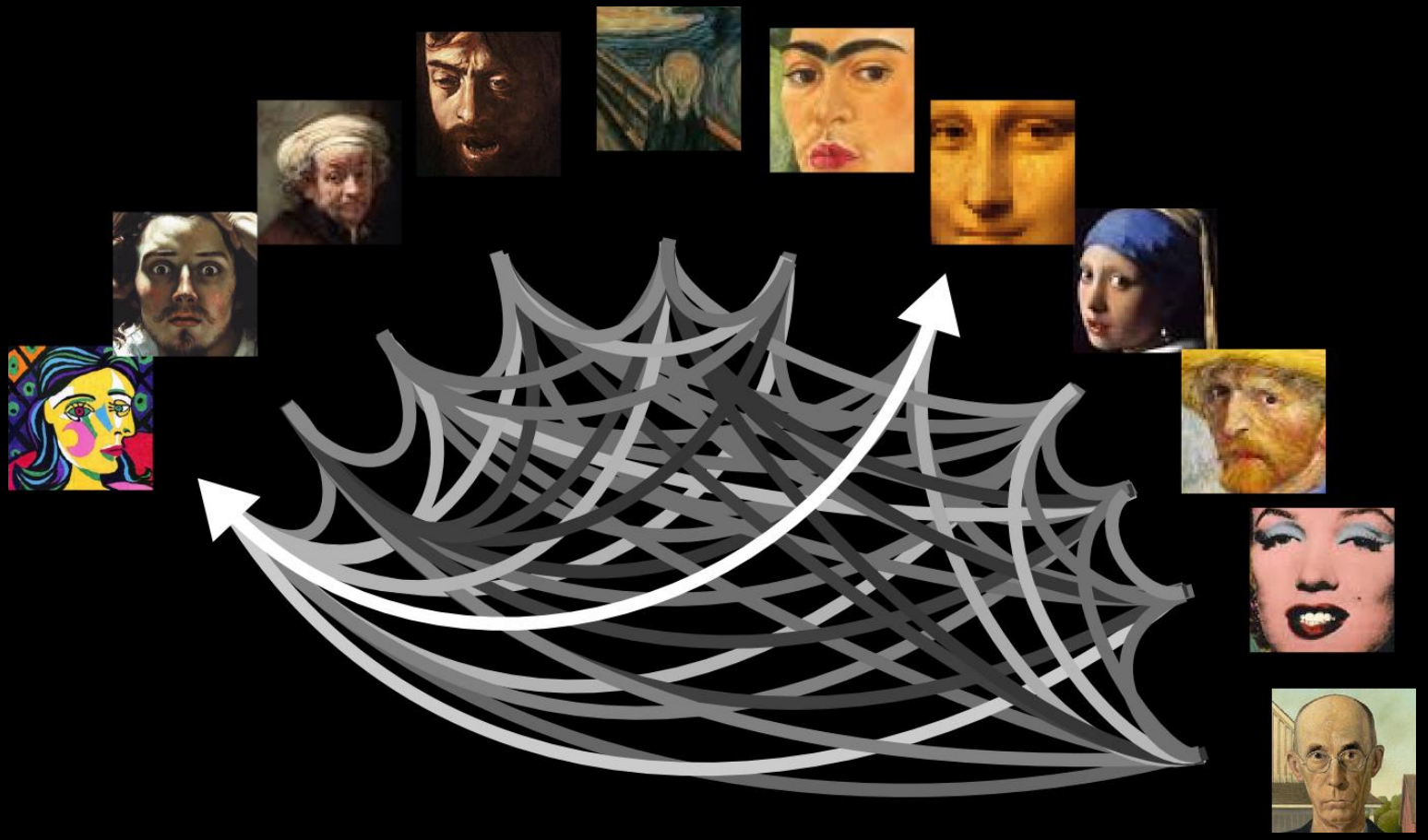
You compare everyone with everyone else

→ **Any match suffices!**

Dedup Est Machina: Birthday Paradox

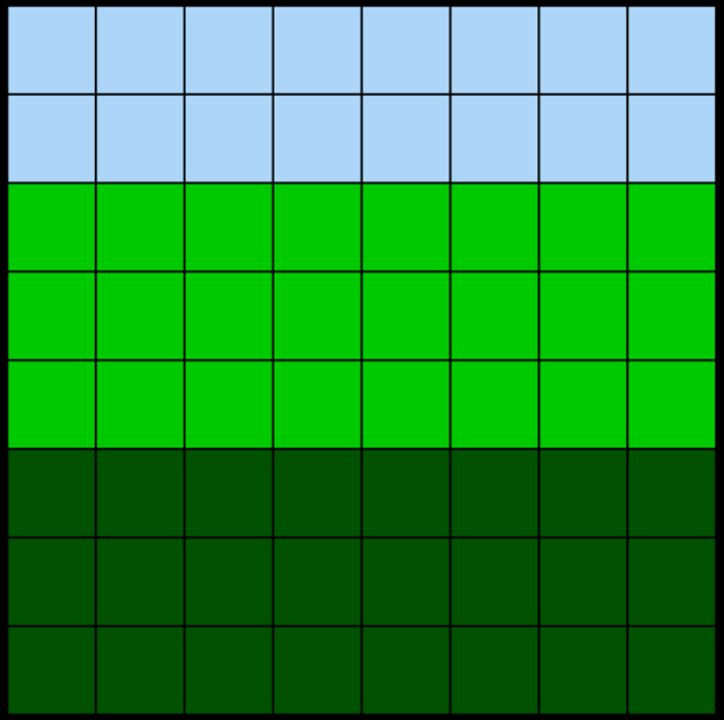


Dedup Est Machina: Birthday Paradox

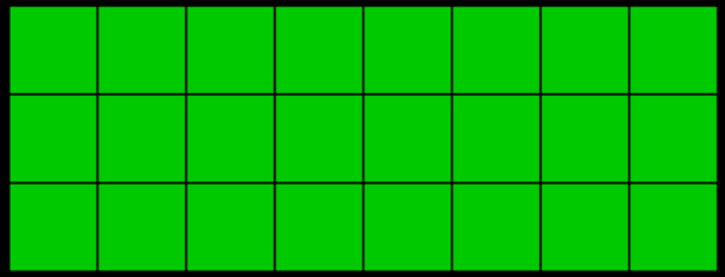


Primitive #3: Birthday Heapspray

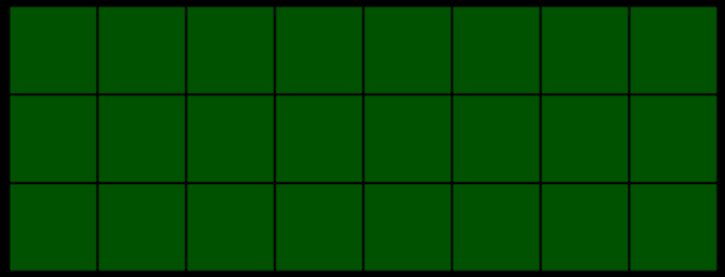
physical memory



attacker memory

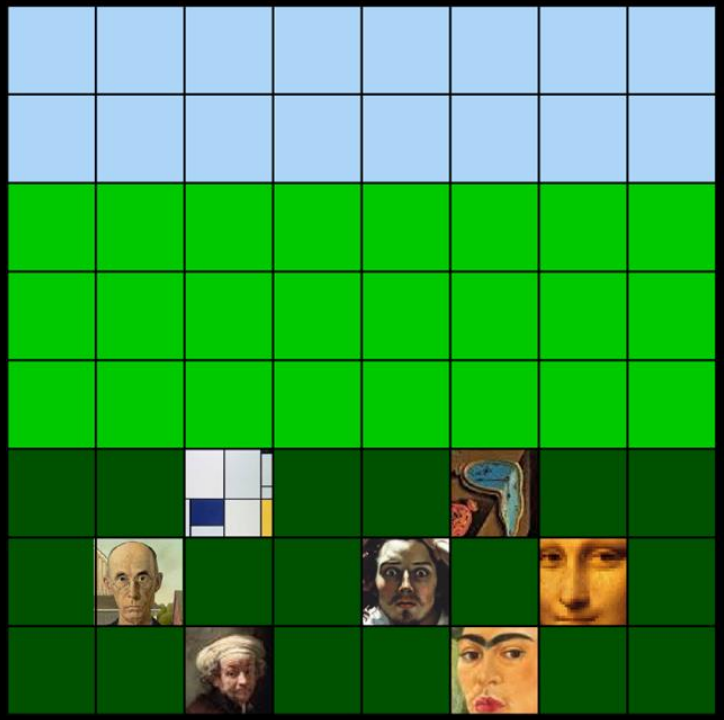


victim memory

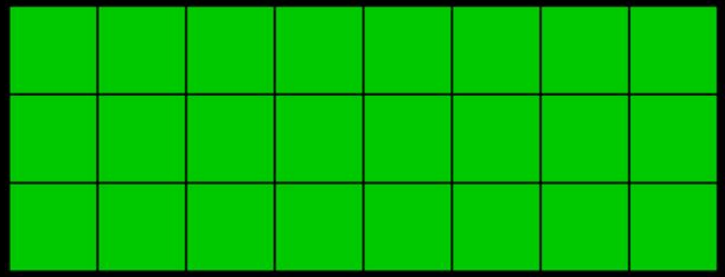


Primitive #3: Birthday Heapspray

physical memory



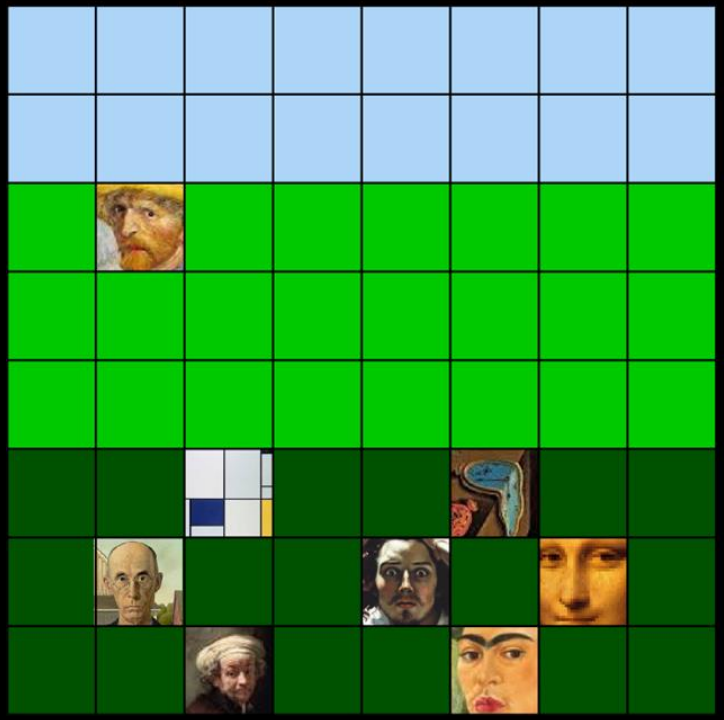
attacker memory



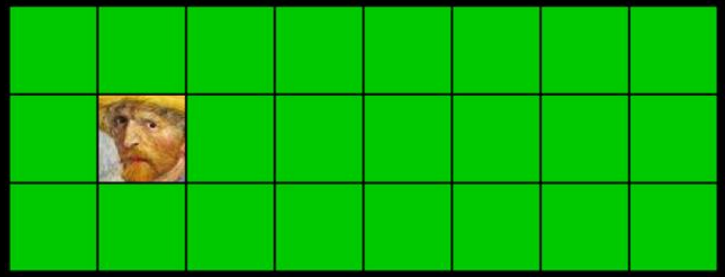
victim memory

Primitive #3: Birthday Heapspray

physical memory



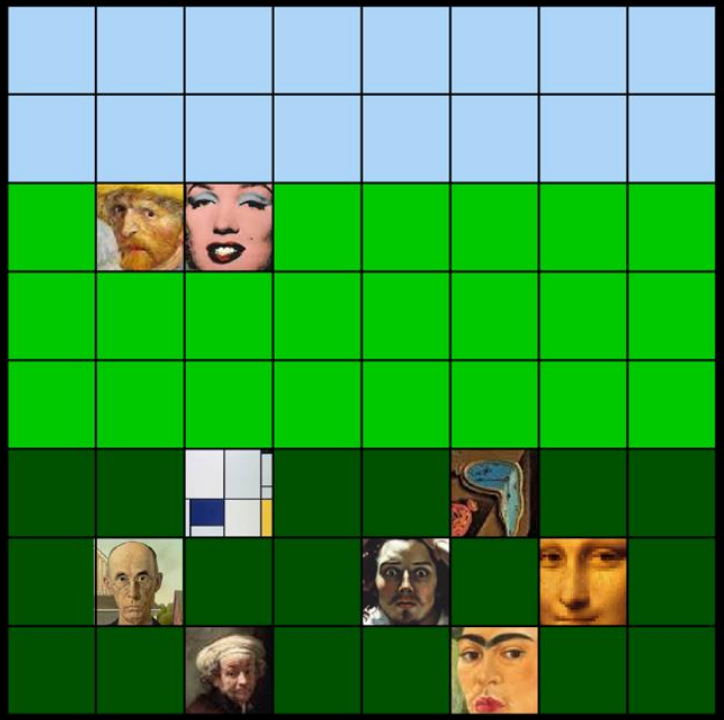
attacker memory



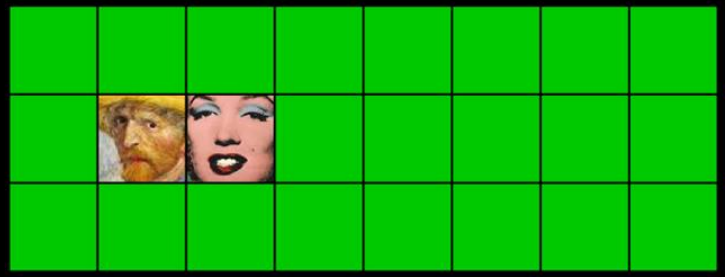
victim memory

Primitive #3: Birthday Heapspray

physical memory



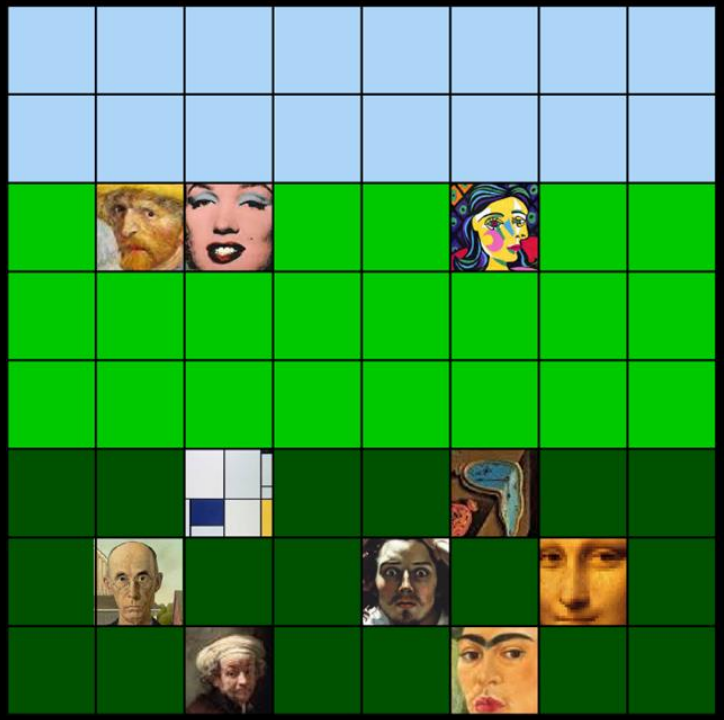
attacker memory



victim memory

Primitive #3: Birthday Heapspray

physical memory

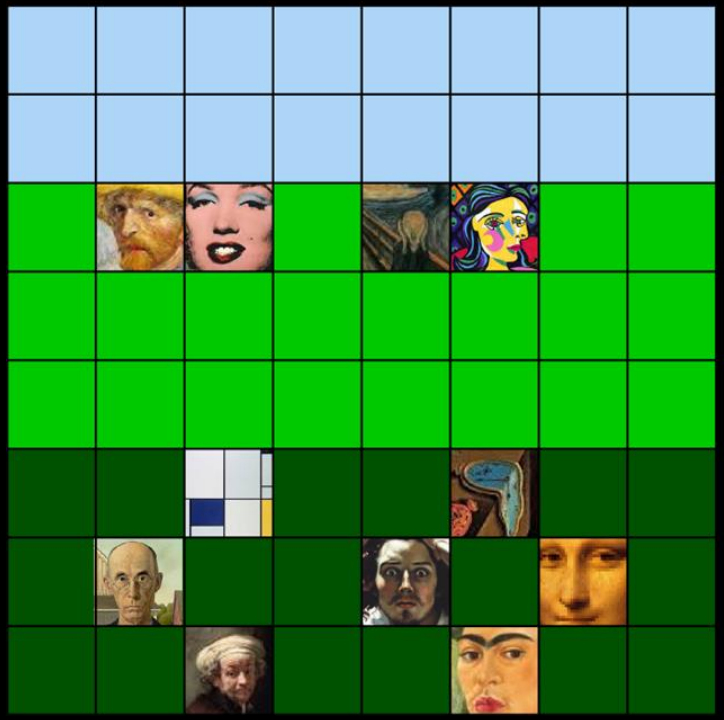


attacker memory

victim memory

Primitive #3: Birthday Heapspray

physical memory

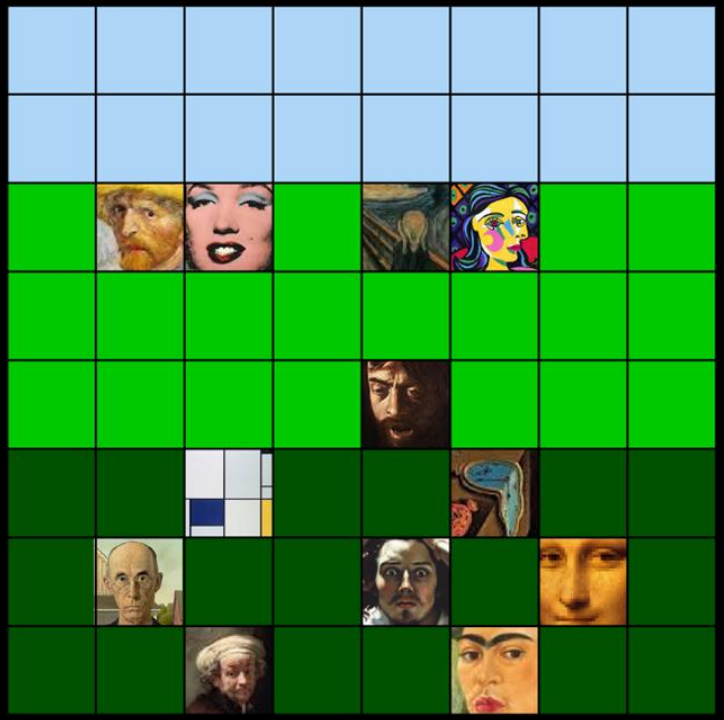


attacker memory

victim memory

Primitive #3: Birthday Heapspray

physical memory

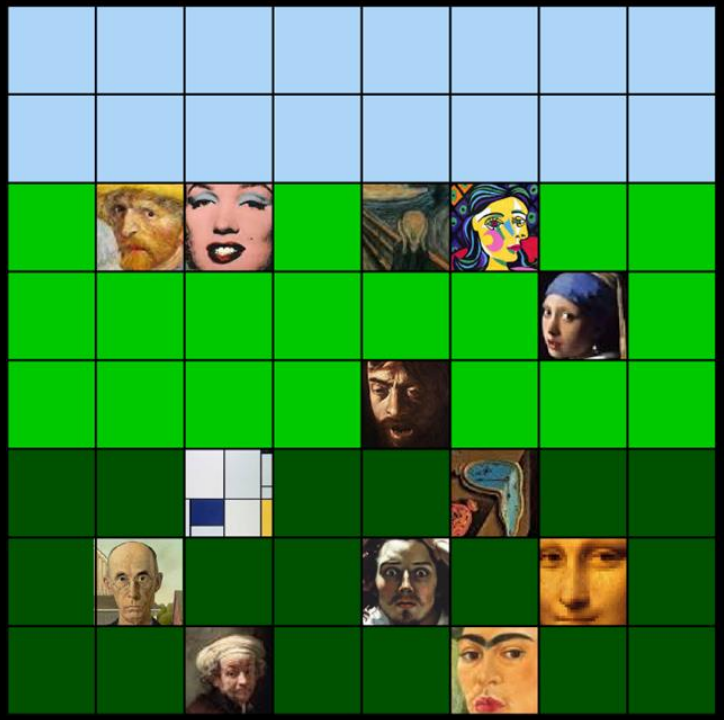


attacker memory

victim memory

Primitive #3: Birthday Heapspray

physical memory

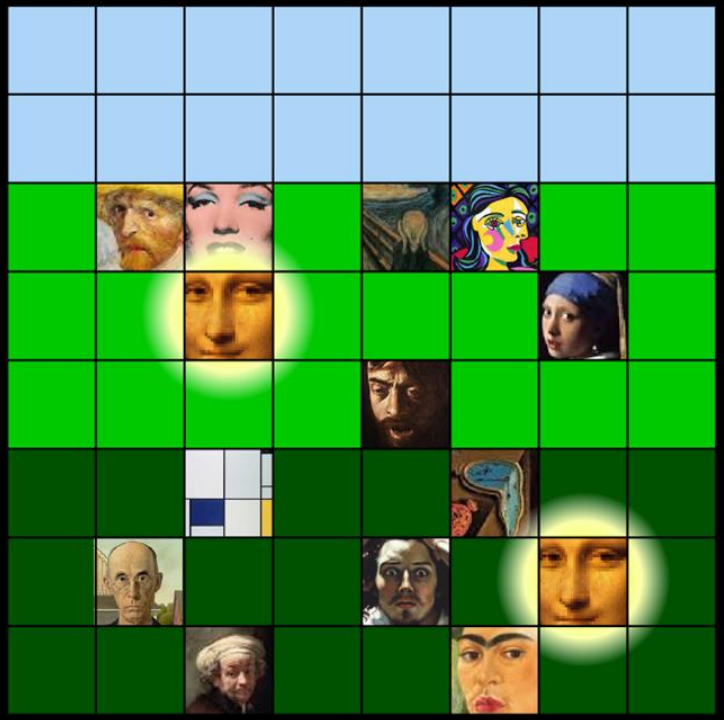


attacker memory

victim memory

Primitive #3: Birthday Heapspray

physical memory



attacker memory

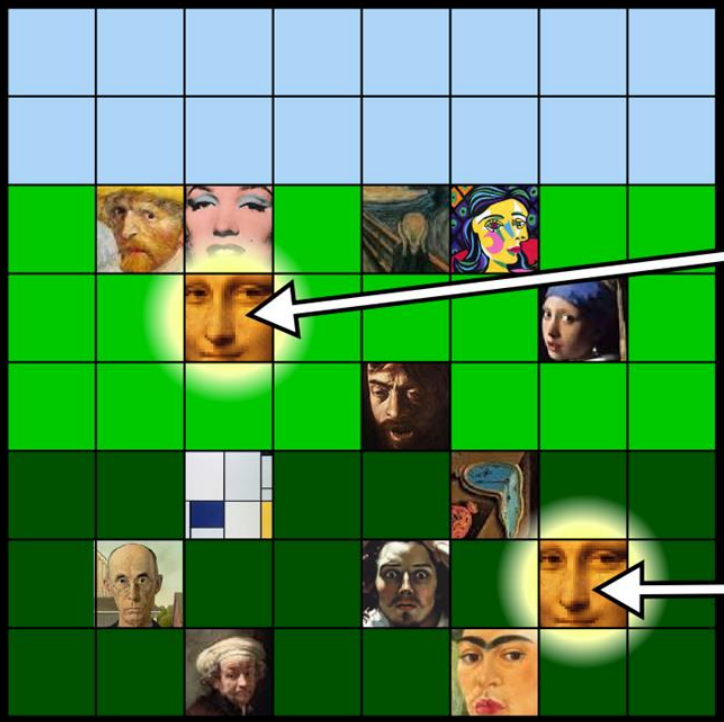
victim memory

Primitive #3: Birthday Heapspray

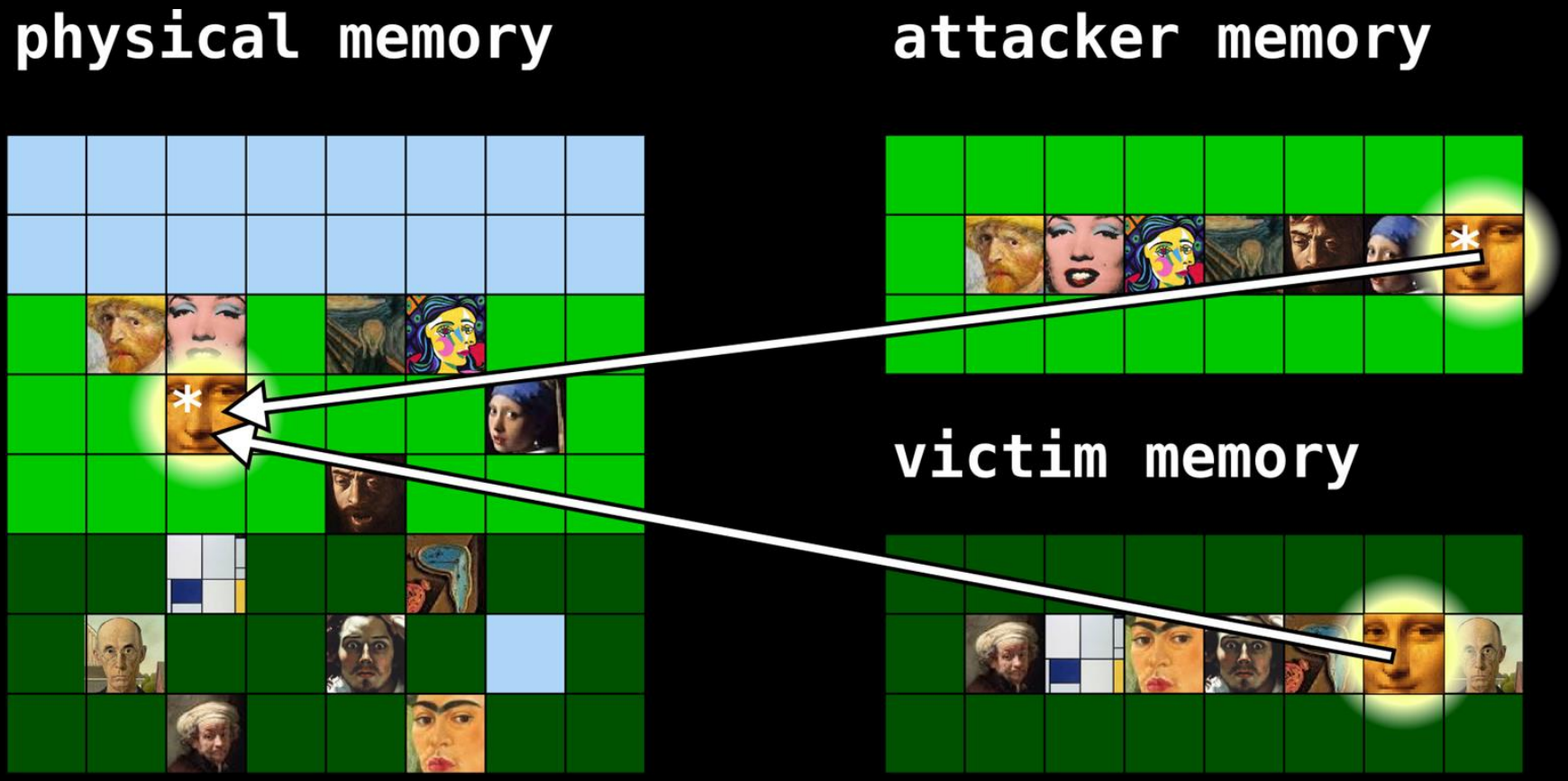
physical memory

attacker memory

victim memory



Primitive #3: Birthday Heapspray

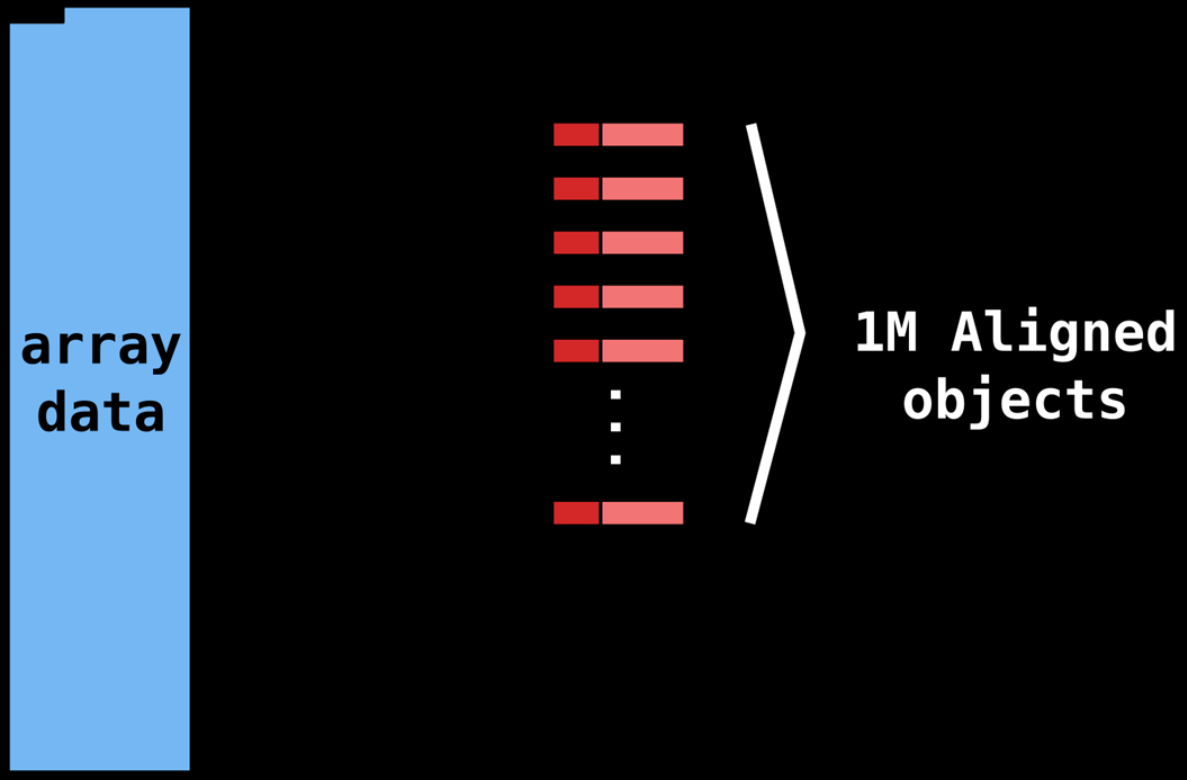


Dedup Est Machina: Leaking Heap Pointer (#3)

Creating Secret Pages

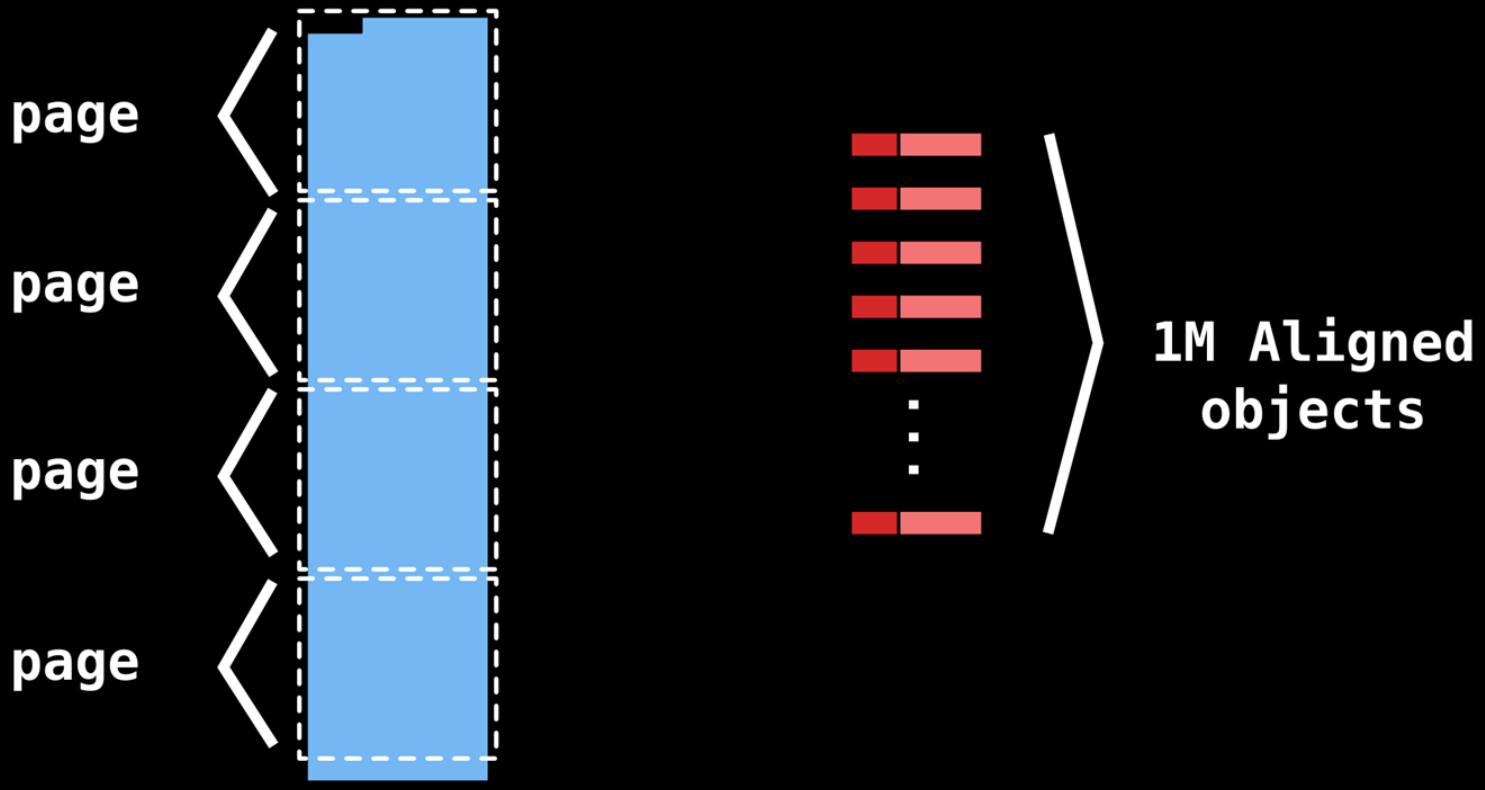
Dedup Est Machina: Leaking Heap Pointer (#3)

Creating Secret Pages



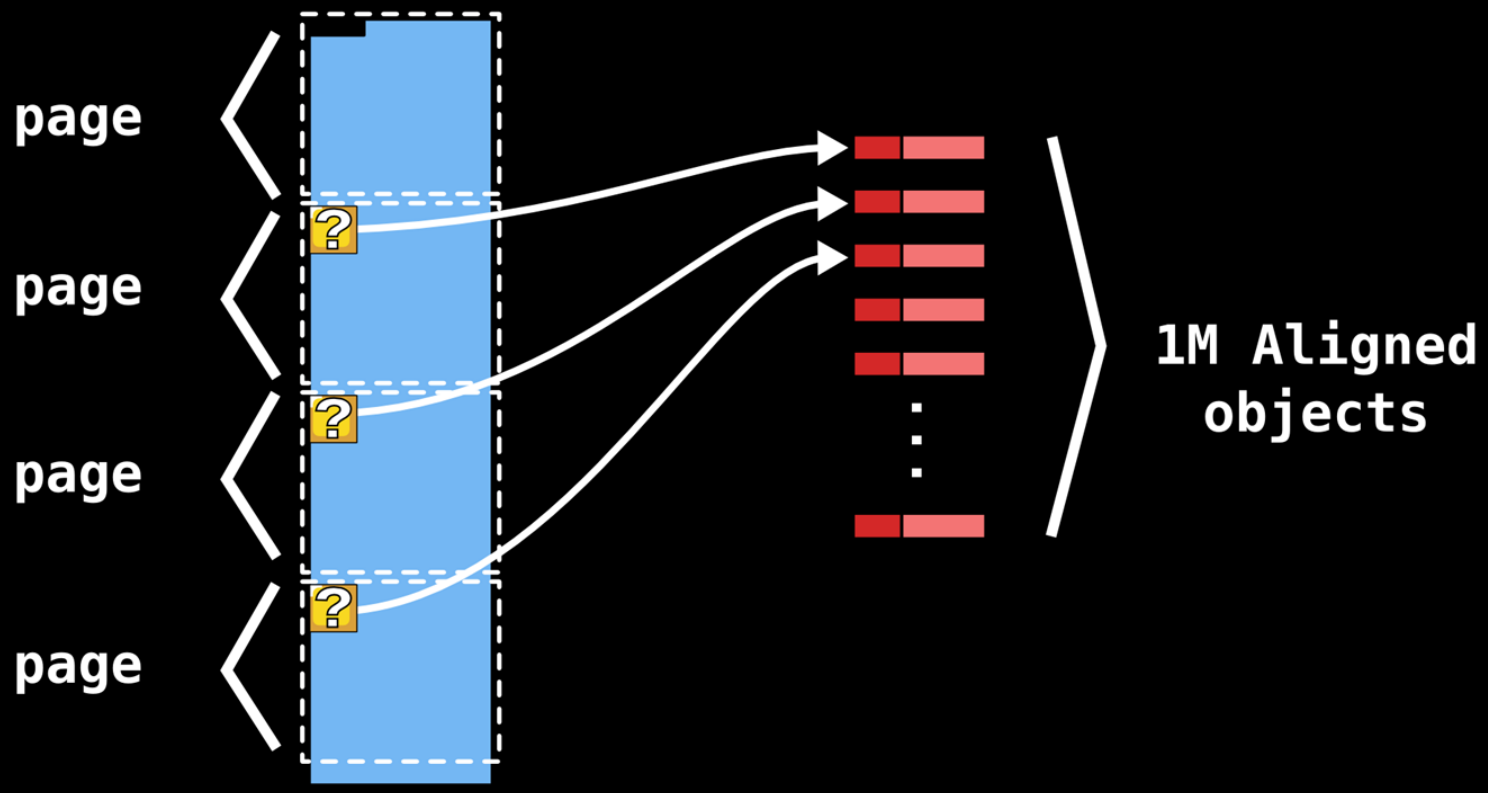
Dedup Est Machina: Leaking Heap Pointer (#3)

Creating Secret Pages



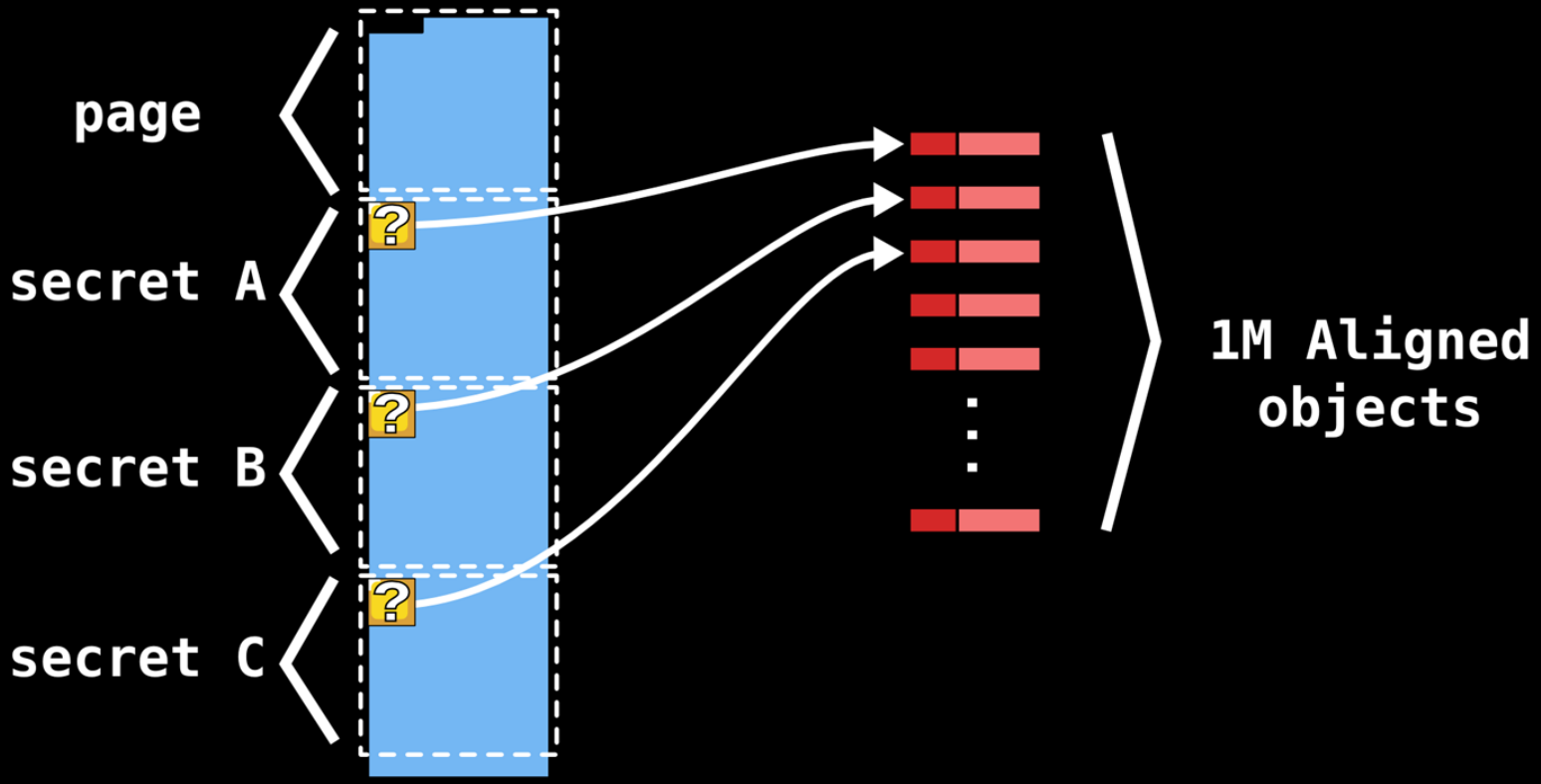
Dedup Est Machina: Leaking Heap Pointer (#3)

Creating Secret Pages



Dedup Est Machina: Leaking Heap Pointer (#3)

Creating Secret Pages

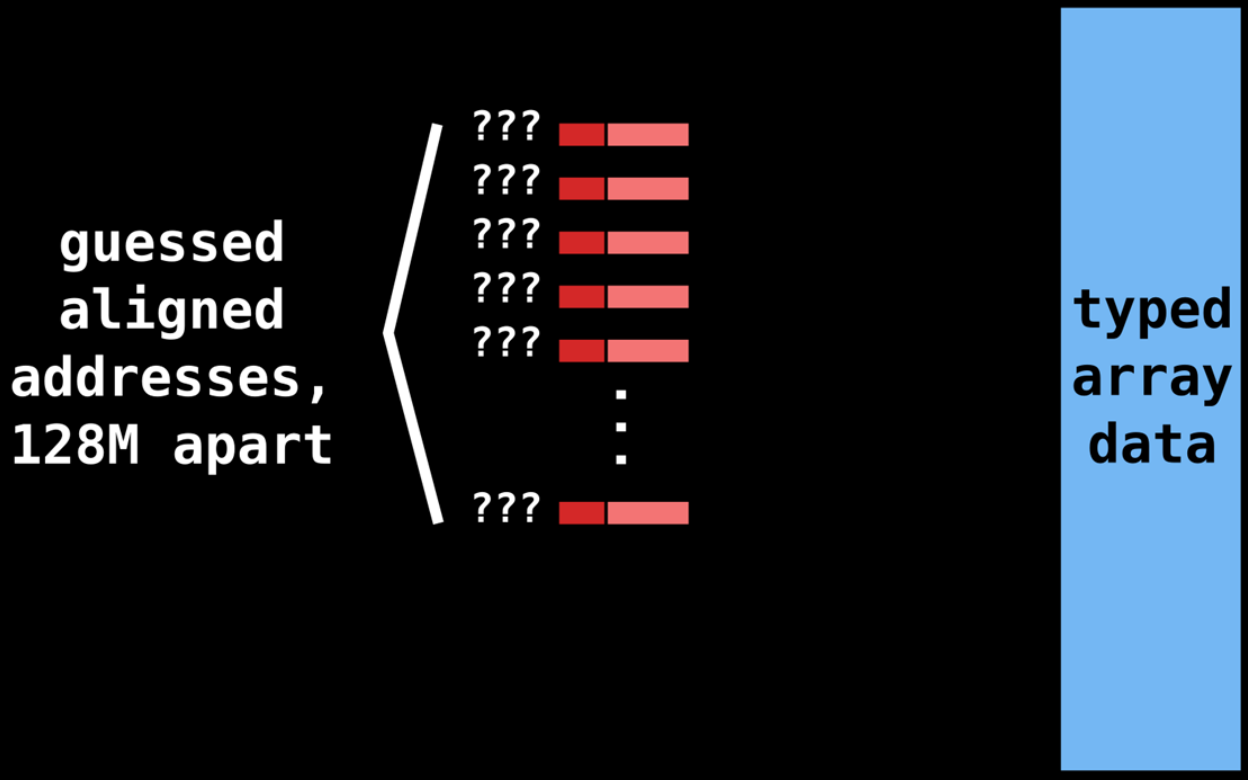


Dedup Est Machina: Leaking Heap Pointer (#3)

Creating Probe Pages

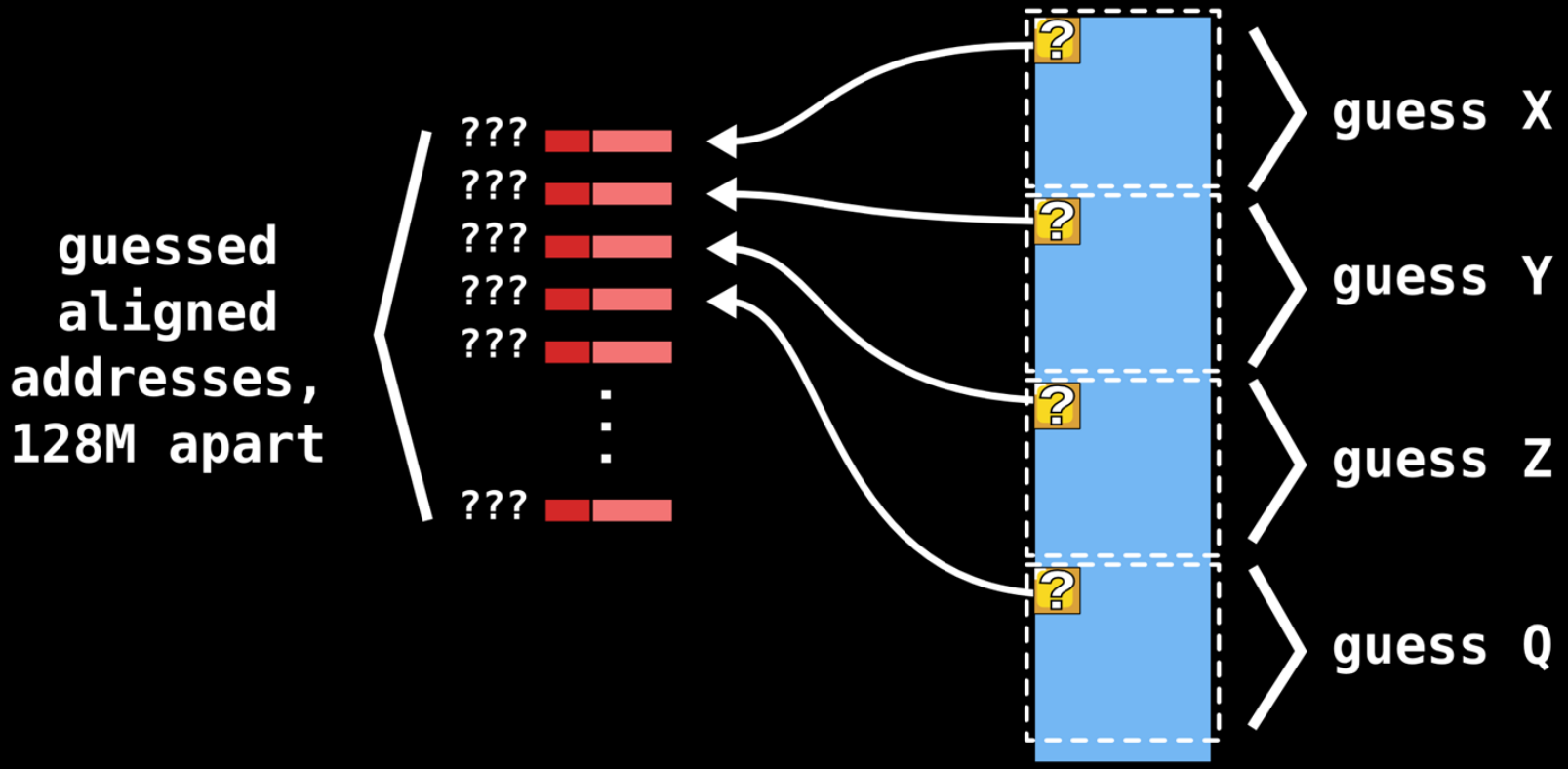
Dedup Est Machina: Leaking Heap Pointer (#3)

Creating Probe Pages



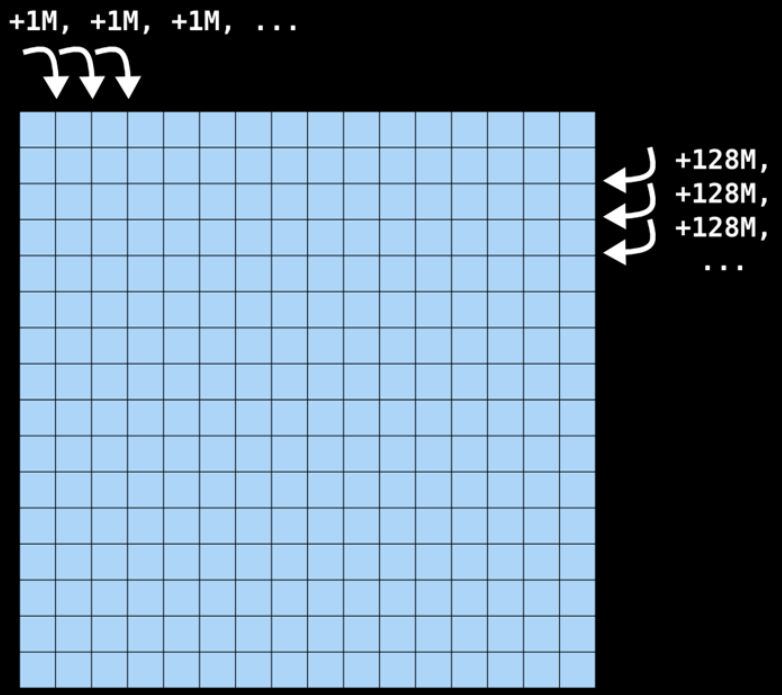
Dedup Est Machina: Leaking Heap Pointer (#3)

Creating Probe Pages



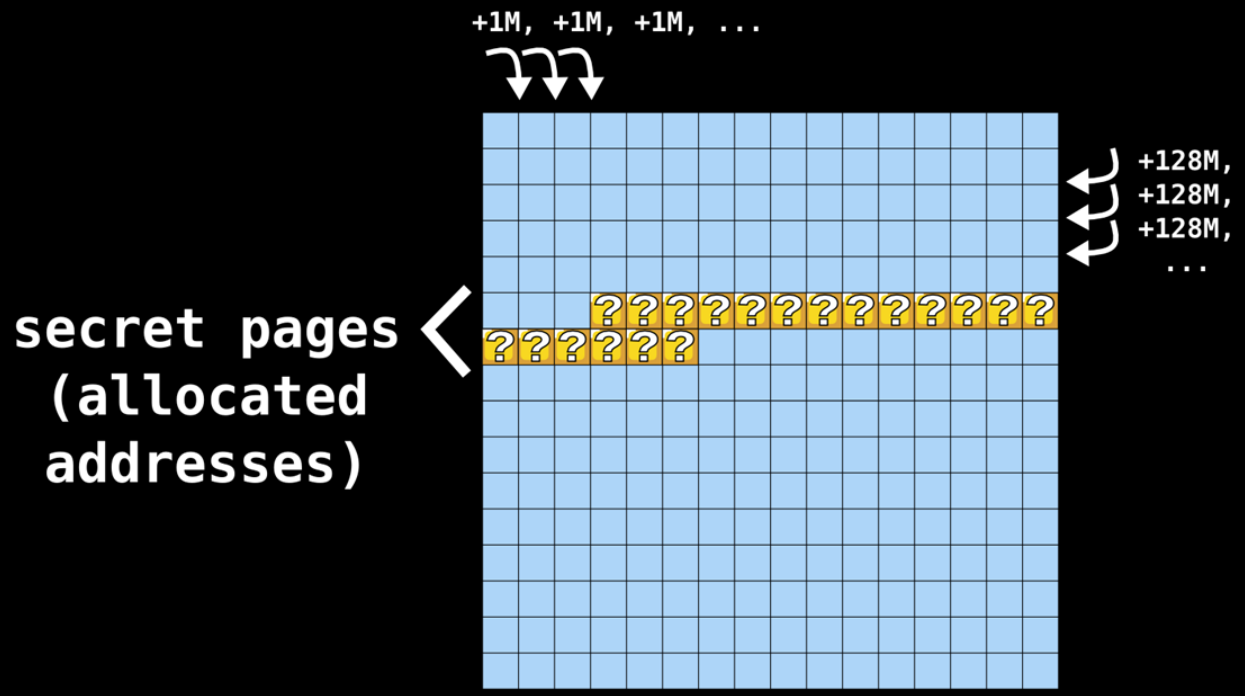
Dedup Est Machina: Leaking Heap Pointer (#3)

Birthday Heapspray



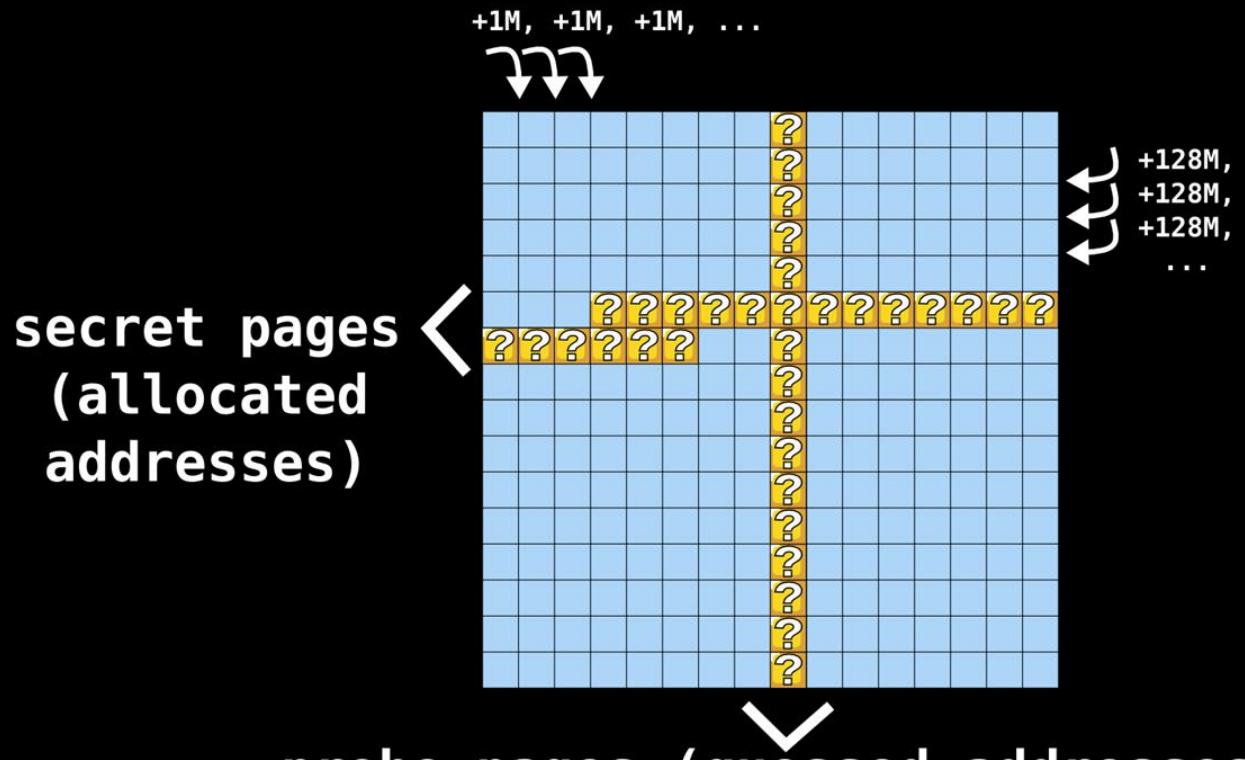
Dedup Est Machina: Leaking Heap Pointer (#3)

Birthday Heapspray



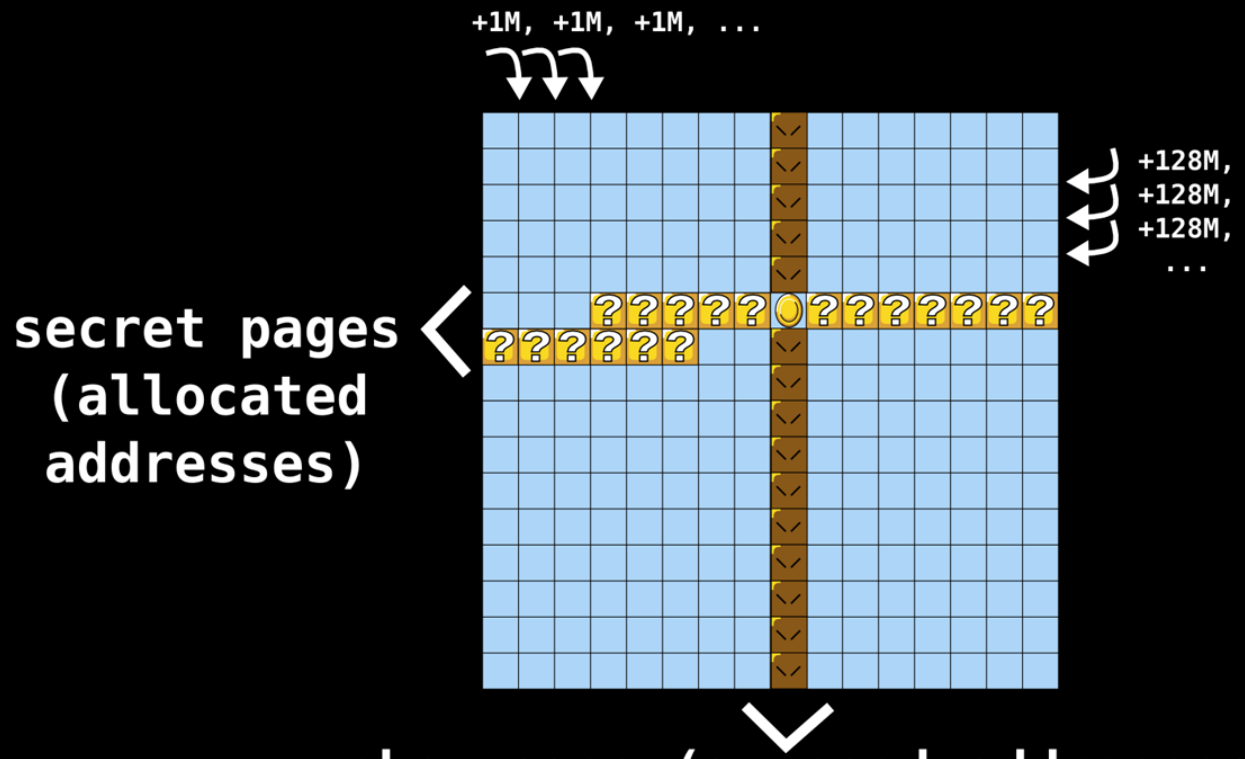
Dedup Est Machina: Leaking Heap Pointer (#3)

Birthday Heapspray



Dedup Est Machina: Leaking Heap Pointer (#3)

Birthday Heapspray

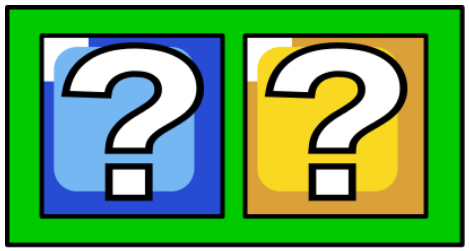


Dedup Est Machina: Overview

Memory deduplication

Leak randomized heap and code pointers

Create a fake JavaScript object

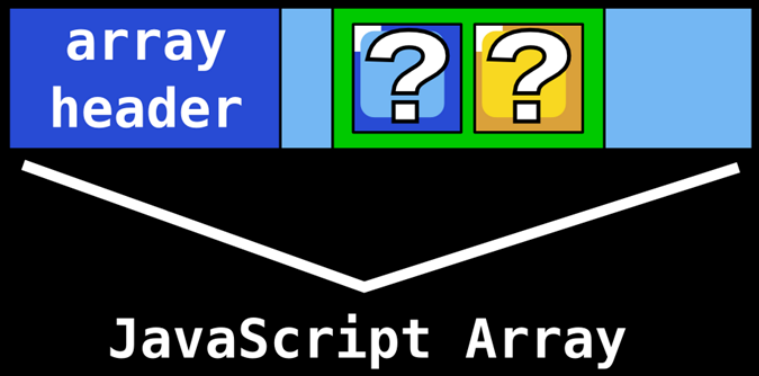


Dedup Est Machina: Creating a Fake Object

Fake JavaScript Uint8Array

Dedup Est Machina: Creating a Fake Object

Fake JavaScript Uint8Array



Dedup Est Machina: Overview

Memory deduplication

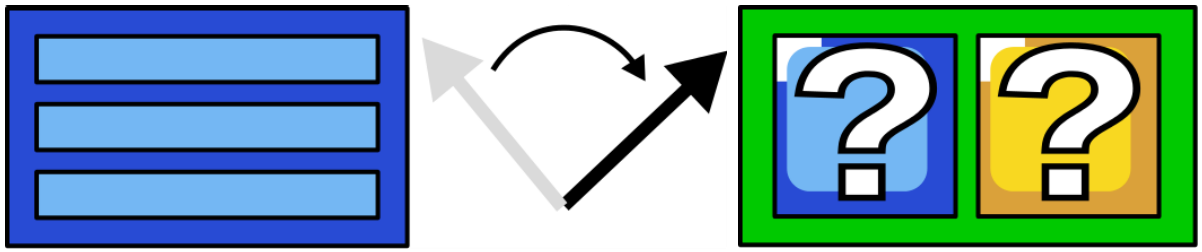
Leak randomized heap and code pointers

Create a fake JavaScript object

+

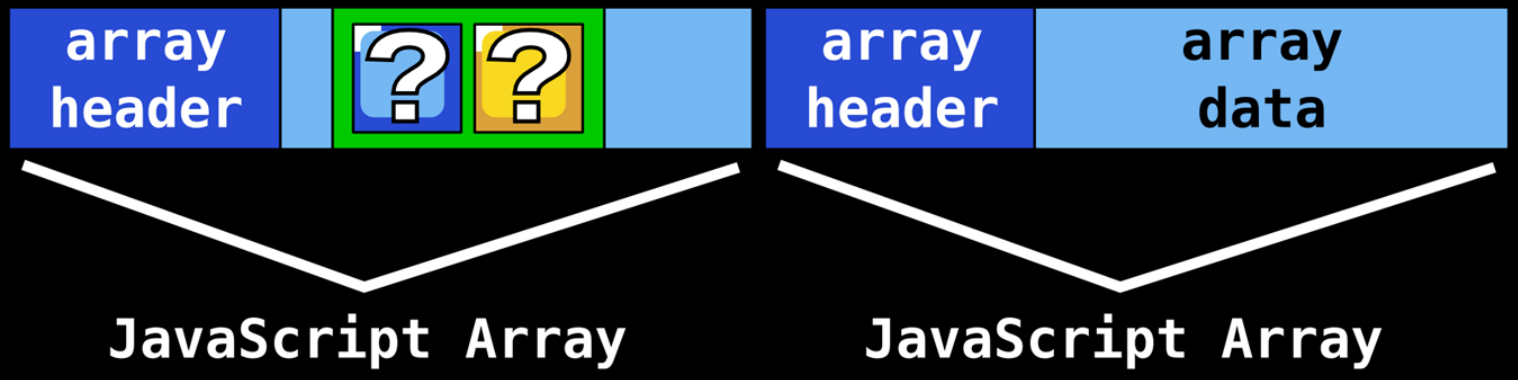
Rowhammer

Create a reference to our fake object



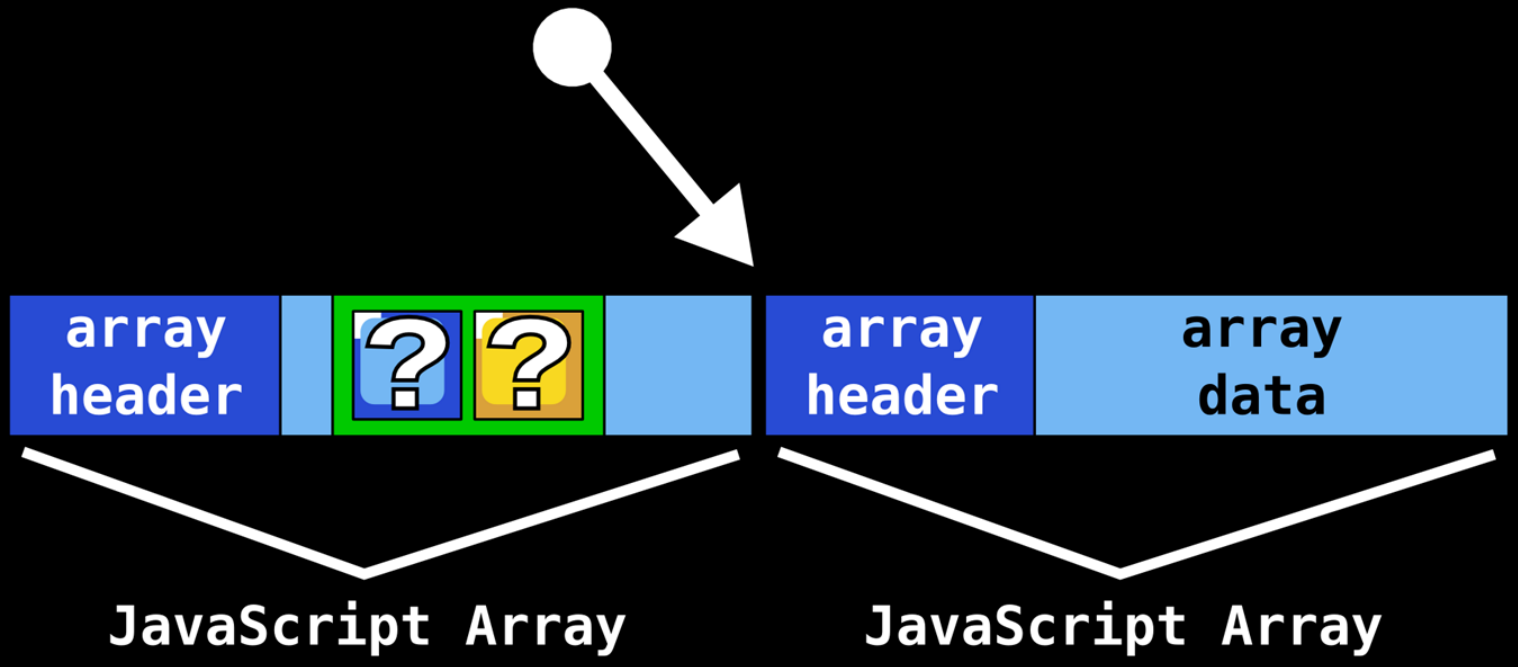
Dedup Est Machina: Creating a Fake Object

Fake JavaScript Uint8Array



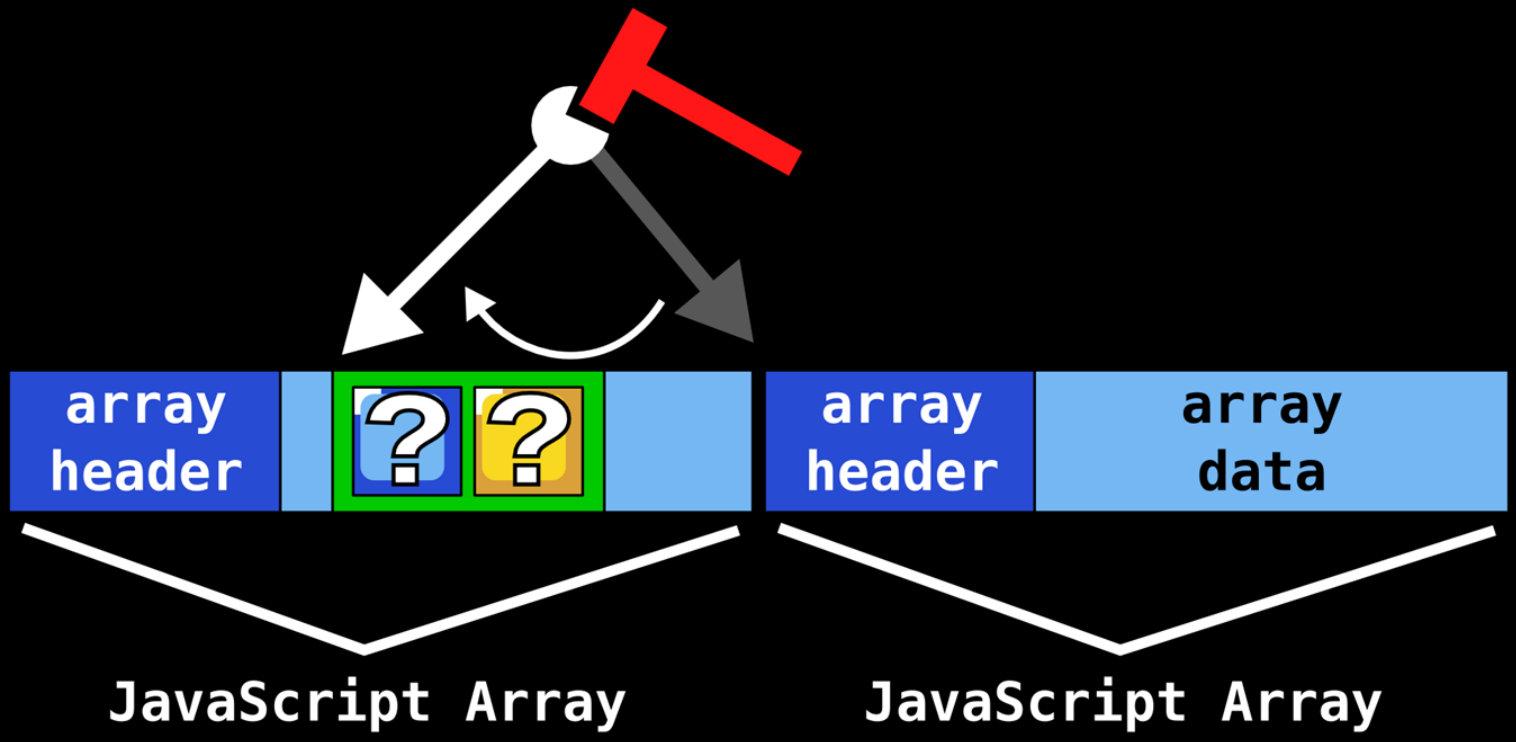
Dedup Est Machina: Creating a Fake Object

Fake JavaScript Uint8Array

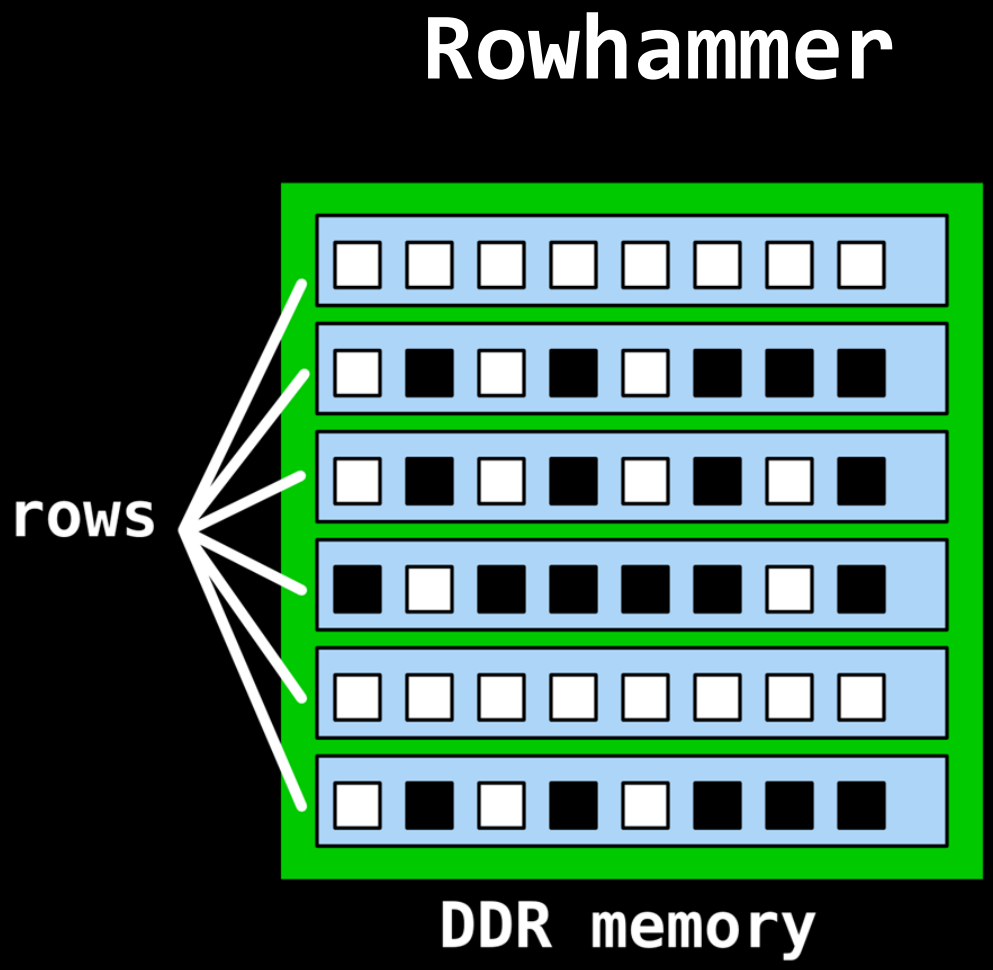


Dedup Est Machina: Creating a Fake Object

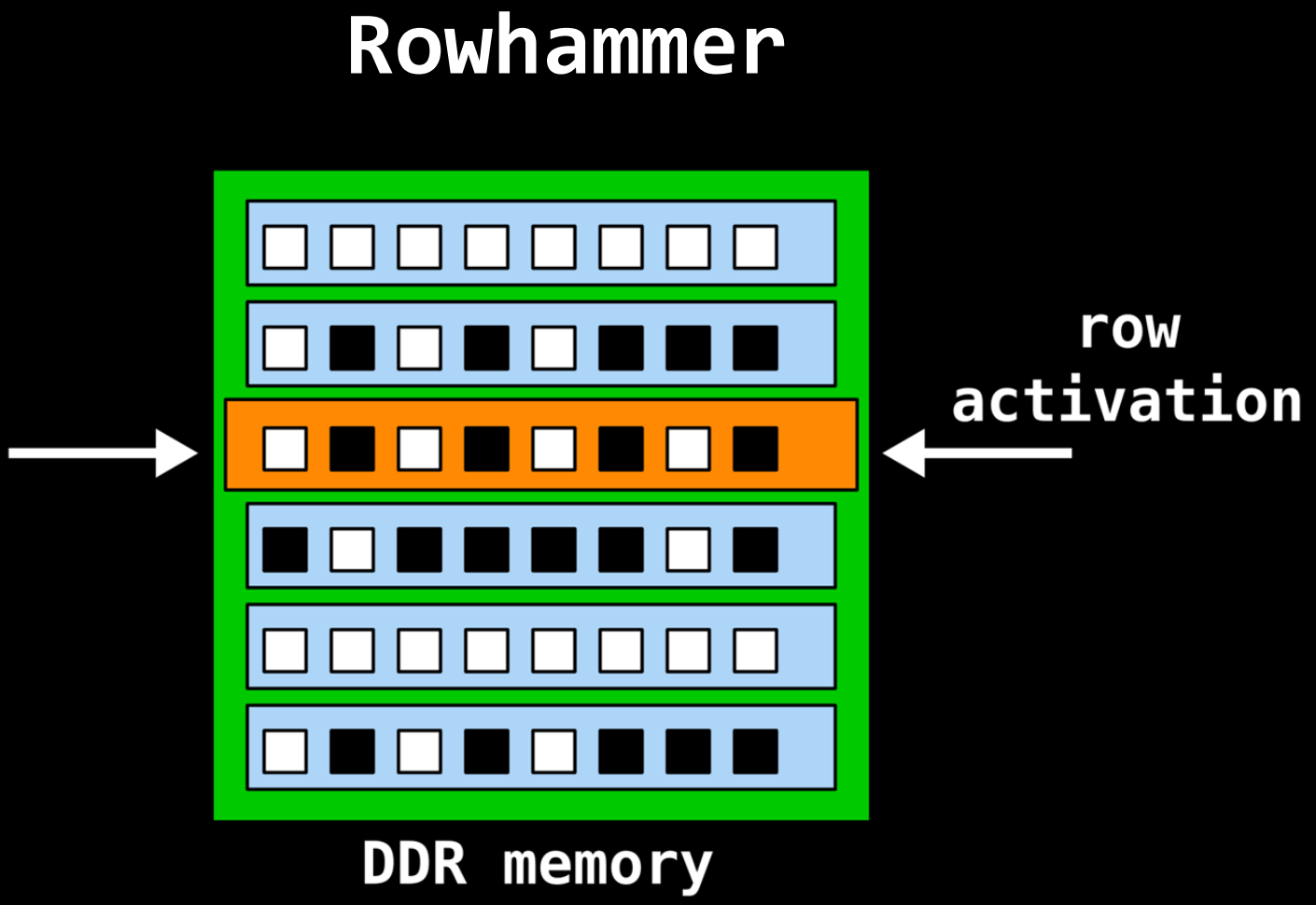
Pointer Pivoting



Dedup Est Machina: Referencing the Fake Object

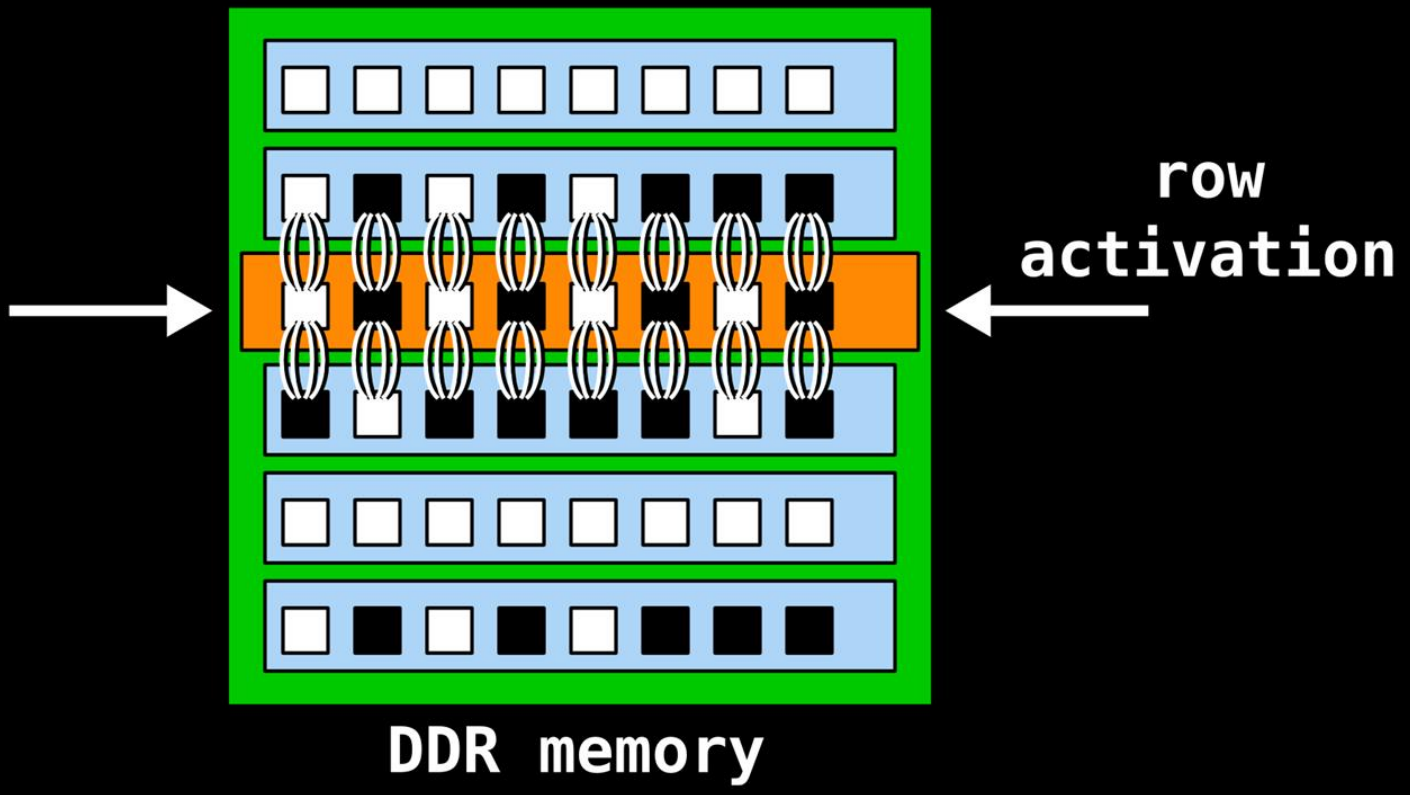


Dedup Est Machina: Referencing the Fake Object



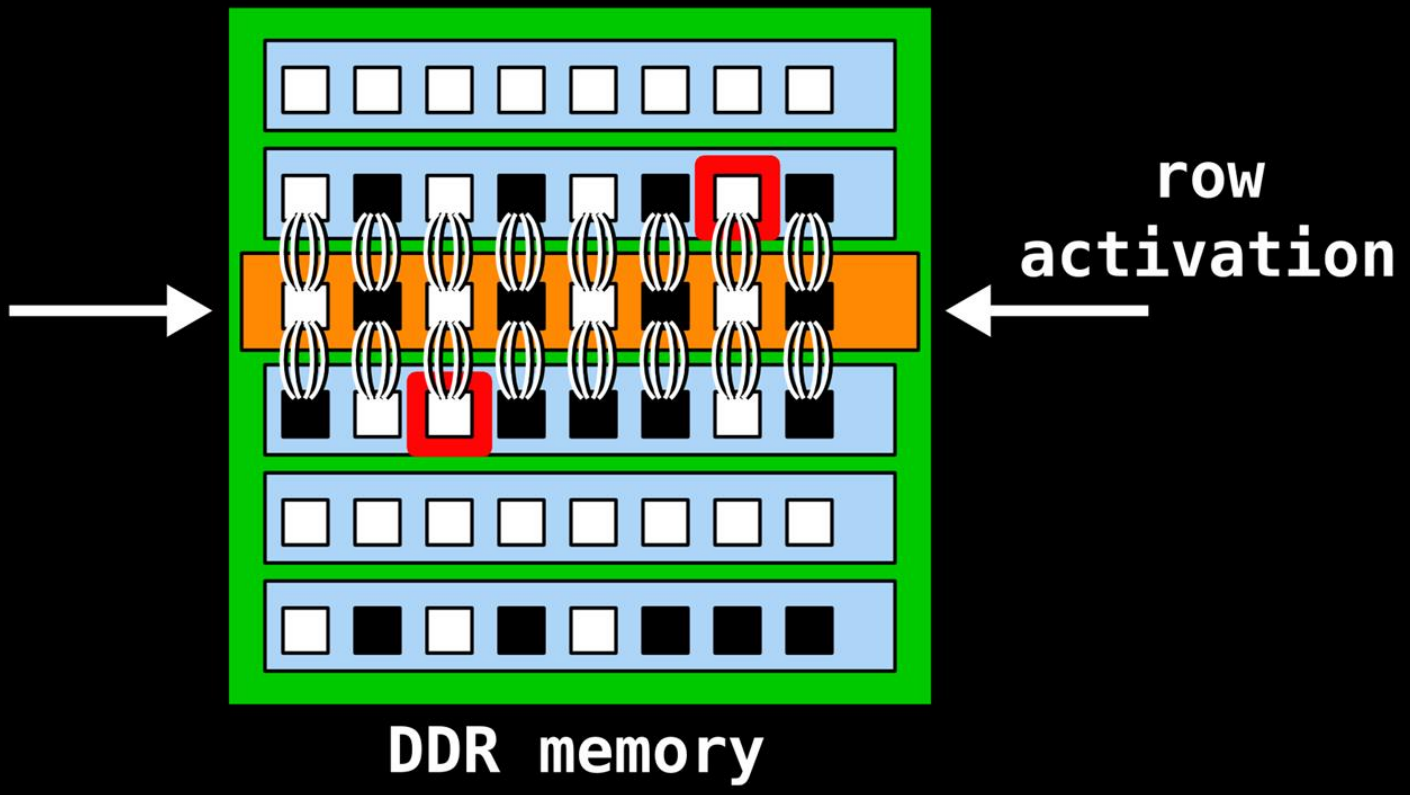
Dedup Est Machina: Referencing the Fake Object

Rowhammer



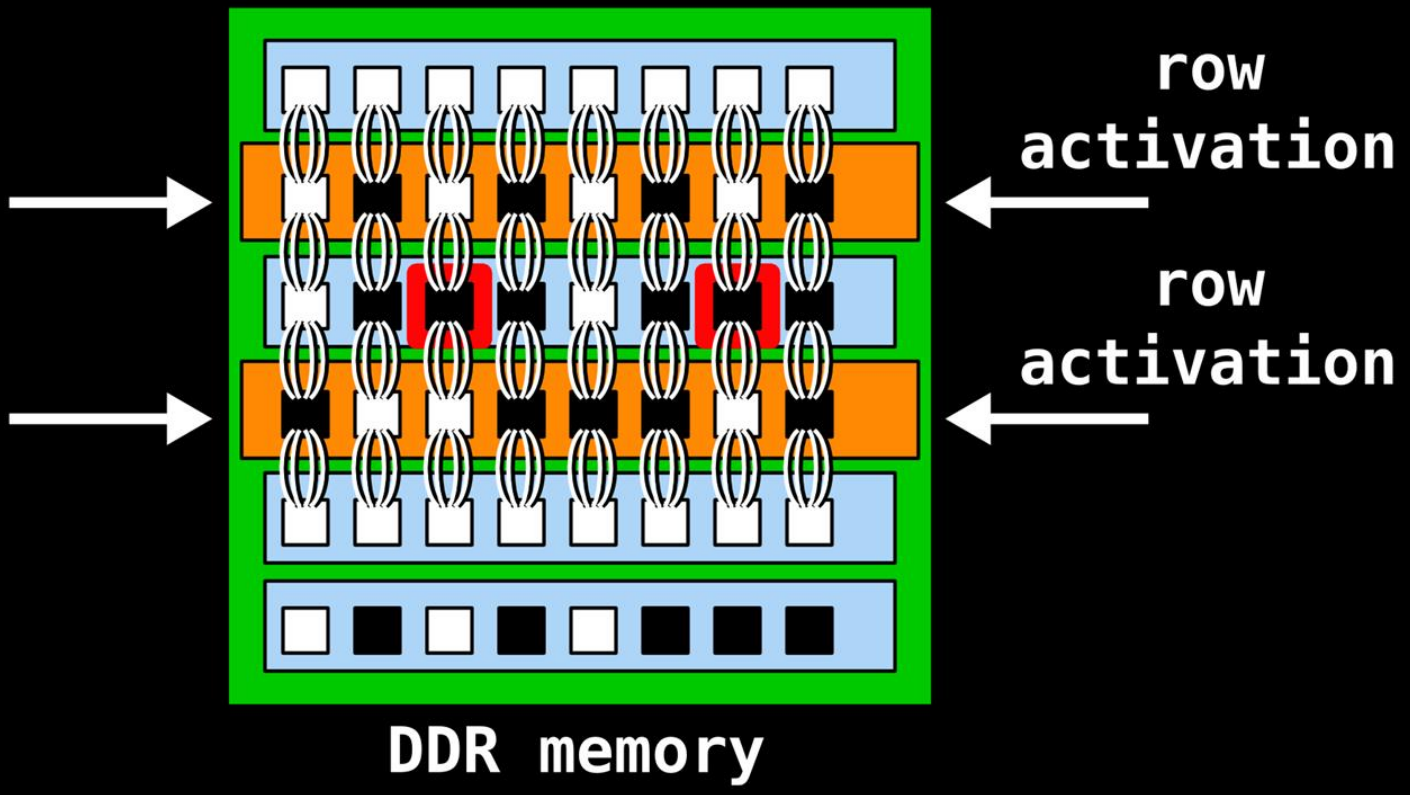
Dedup Est Machina: Referencing the Fake Object

Rowhammer



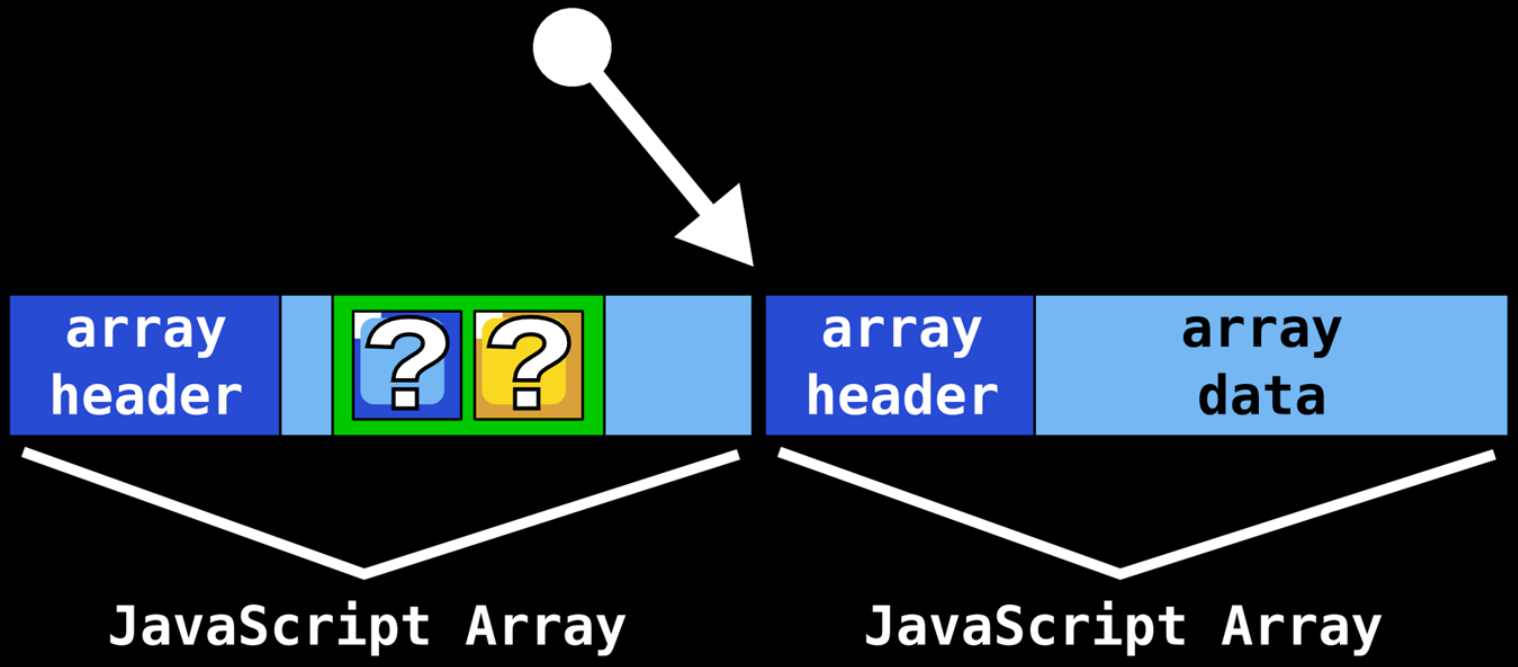
Dedup Est Machina: Referencing the Fake Object

Double-sided Rowhammer



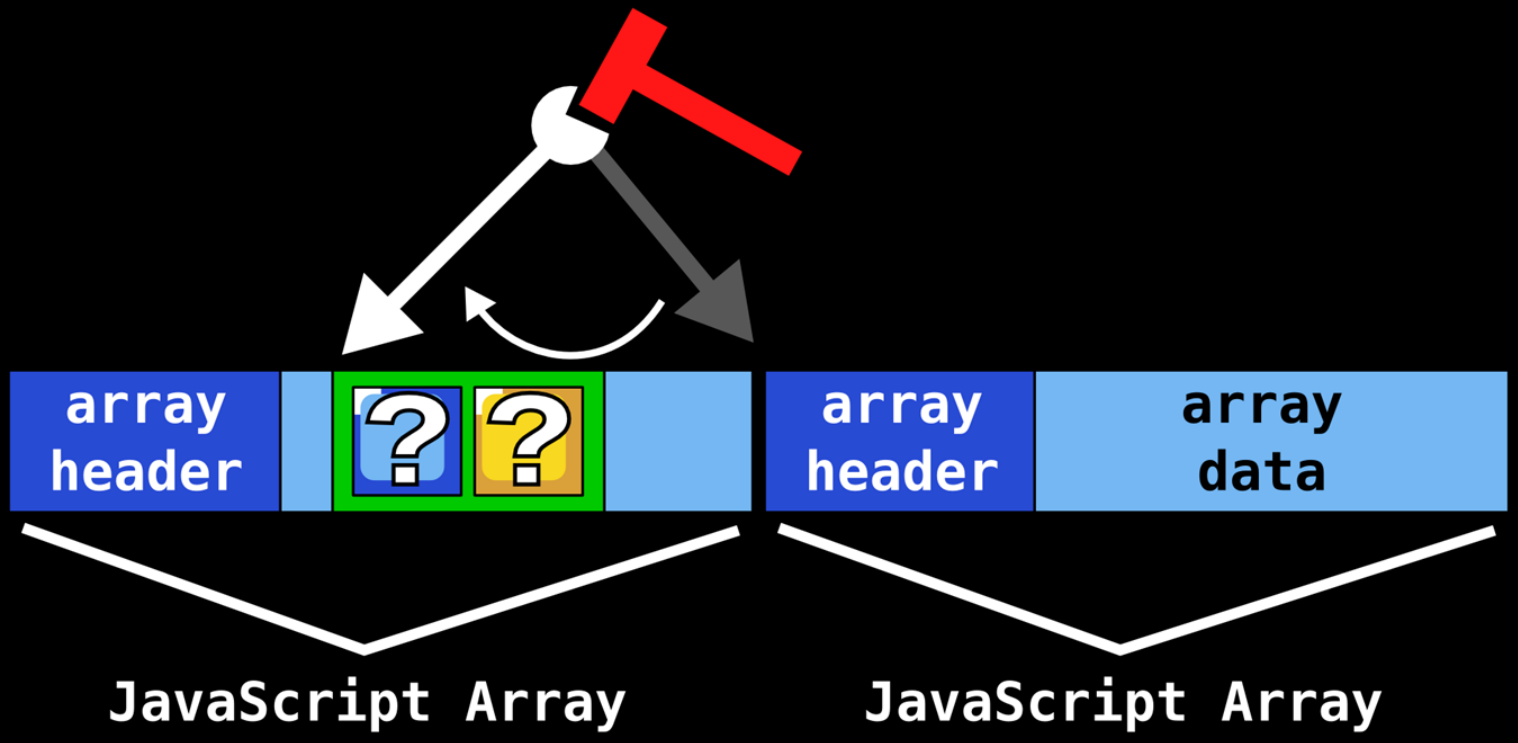
Dedup Est Machina: Referencing the Fake Object

Pointer Pivoting



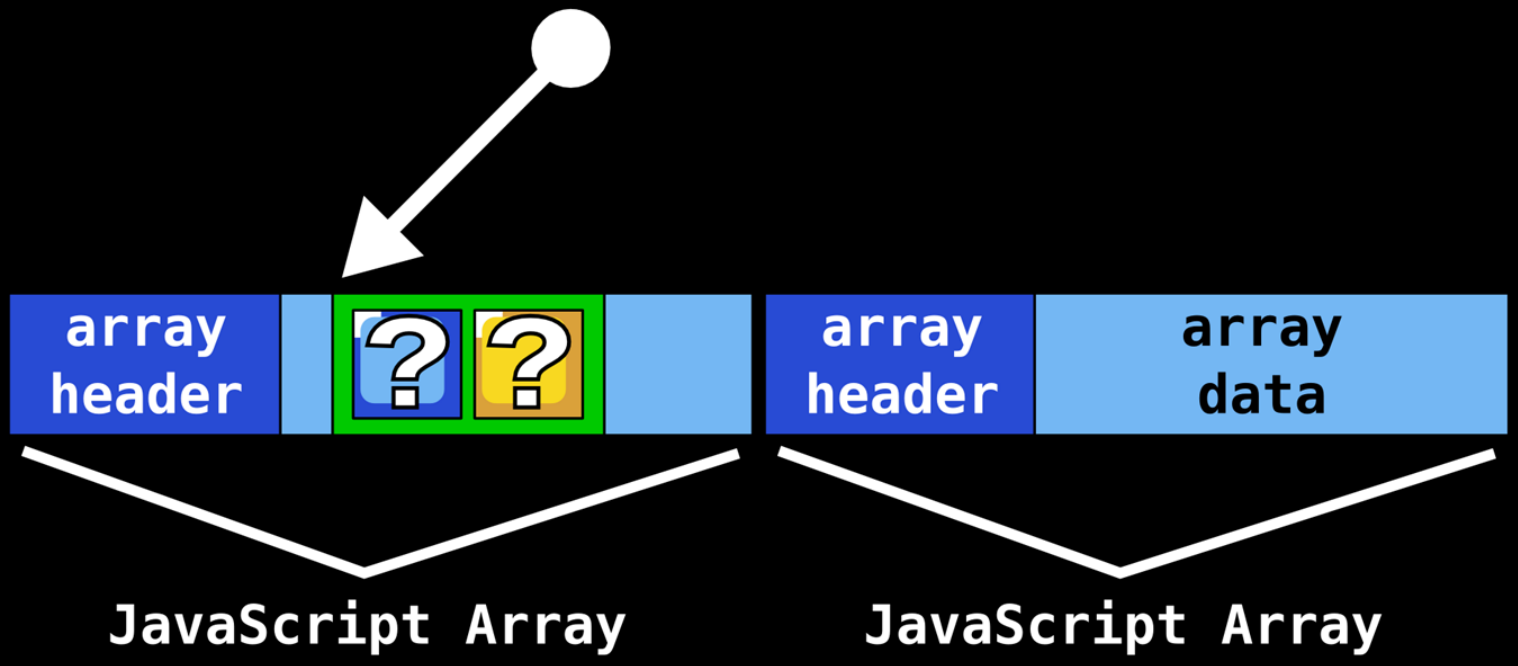
Dedup Est Machina: Referencing the Fake Object

Pointer Pivoting



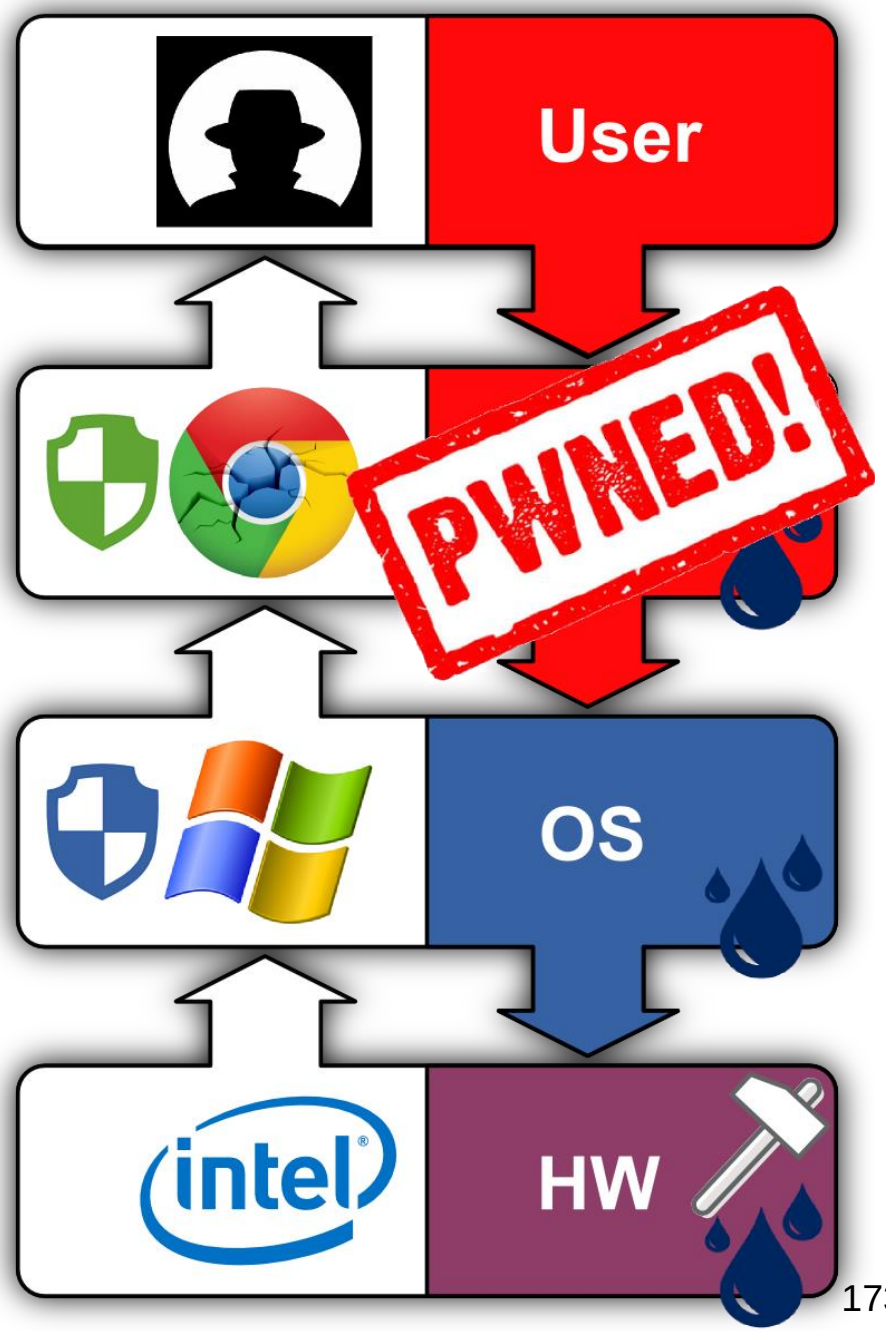
Dedup Est Machina: Referencing the Fake Object

Pointer Pivoting



Dedup Est Machina:

Can One Attack the Full System?



Dedup Est Machina: System-wide Exploitation

Deduplication is enabled system-wide

We can leak secrets from other processes

Say arbitrarily long passwords

- E.g., 30-byte password hashes in nginx

System-wide Rowhammer is more involved

- We don’t “own” other processes’ physical memory

We’ll look at this in our **next example**

Dedup Est Machina: Impact

We shared our MS Edge exploit with Microsoft and they addressed it in MS-16-093, July 18th (CVE-2016-3272) by temporarily disabling memory deduplication on Windows 10

Disable it on legacy systems (Powershell):

```
> Disable-MMAgent -PageCombining
```

Only the beginning

New attack on phones

Can we trust billions of devices?

New attack on VMs

Can we trust the cloud?

EXAMPLE 3

Bug-free Exploitation on Phones

Drammer

Published at CCS 2016 https://www.vusec.net/projects/drammer/

Winner of the Dutch Cyber Security Research Paper Award, 2017

Unprivileged app gets root on Android phones

not a single software

EXAMPLE 4

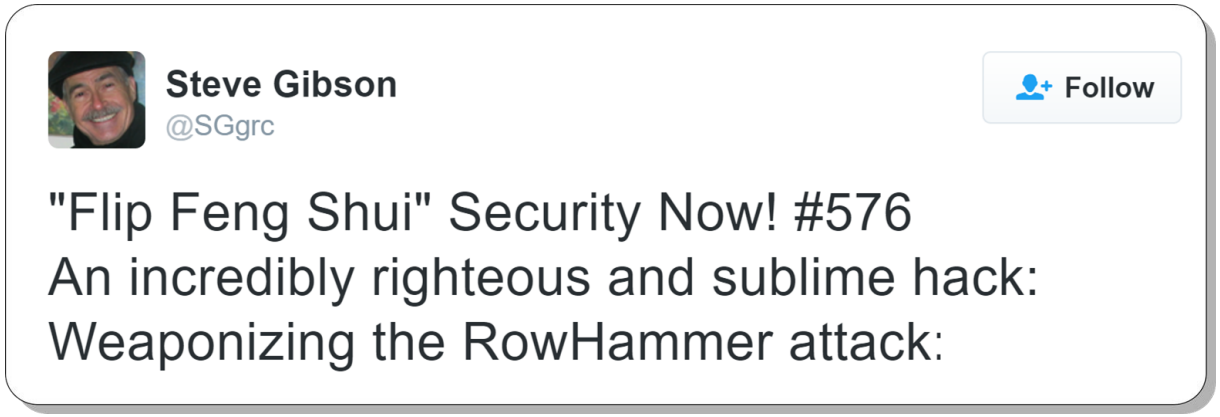
Bug-free Exploitation in Clouds

Flip Feng Shui

Published at USENIX Security 2016

with Ben, Kaveh, Erik, Herbert, and Bart (KU Leuven)

Much media attention



System-wide exploits in public KVM clouds

...without relying on a single software bug

Flip Feng Shui: Overview

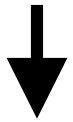
Rowhammer
(hardware glitch)

Flip Feng Shui: Overview

Rowhammer
(hardware glitch)
+
Memory deduplication
(physical memory massaging primitive)

Flip Feng Shui: Overview

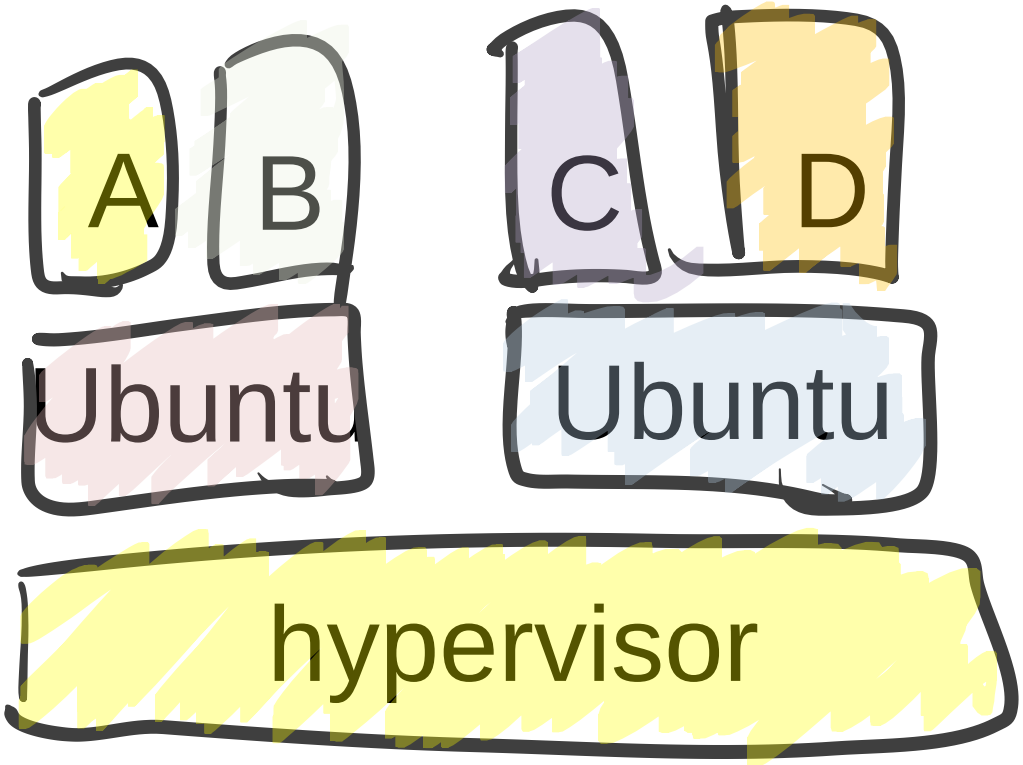
Rowhammer
(hardware glitch)
+
Memory deduplication
(physical memory massaging primitive)



Cross-VM compromise in public Linux/KVM clouds without software bugs

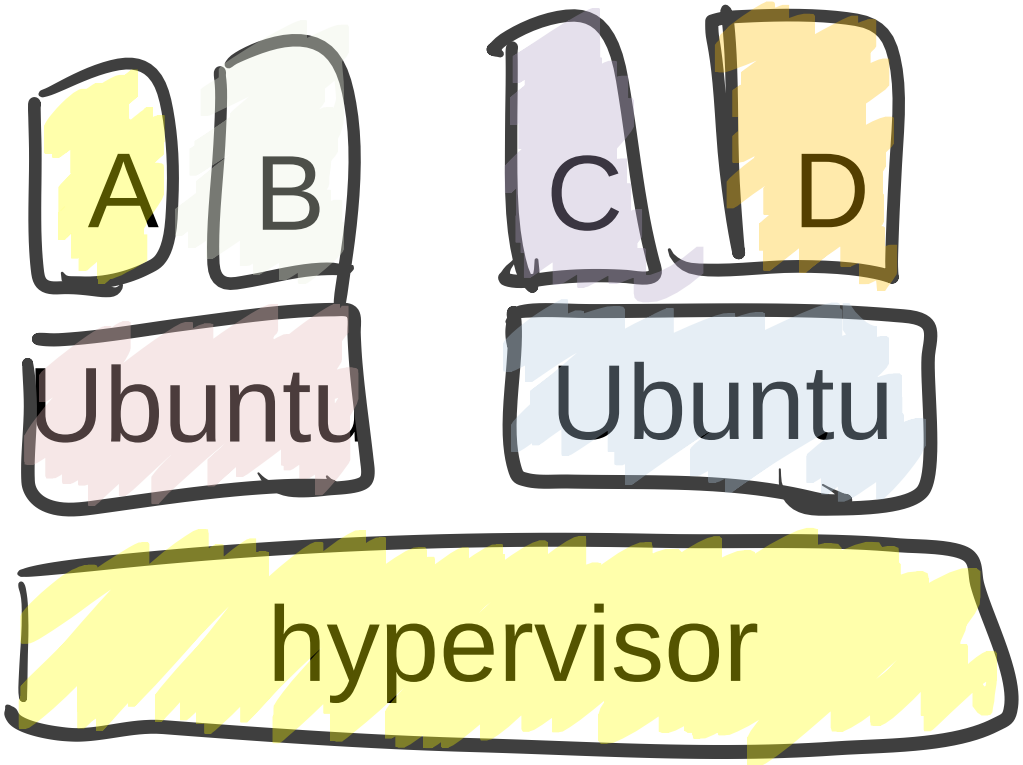
KVM / Clouds

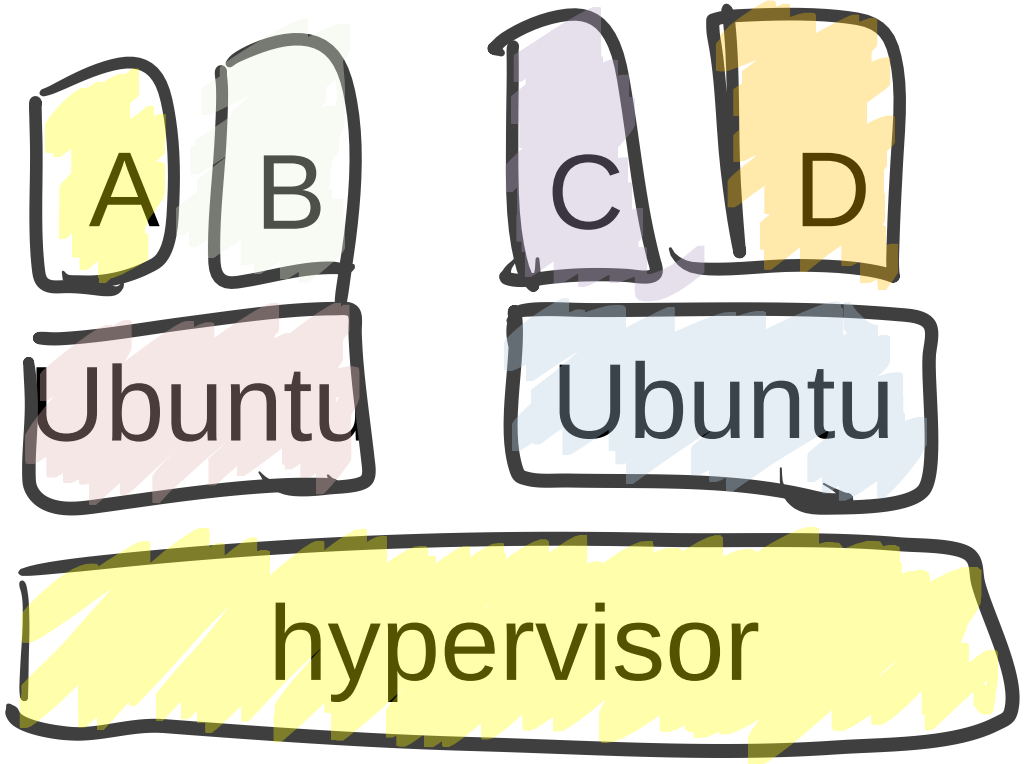
KSM: Kernel Same-page Merging

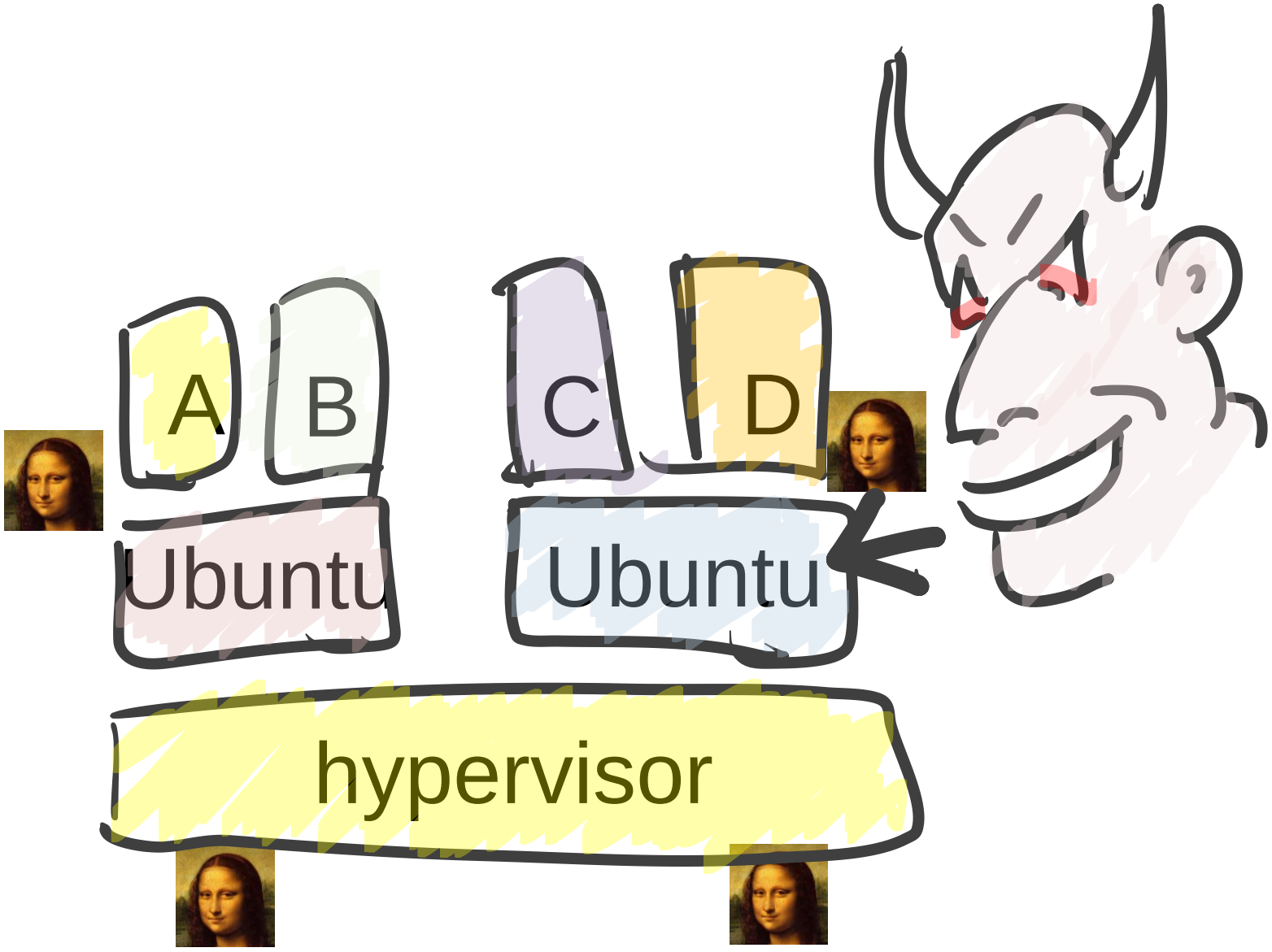


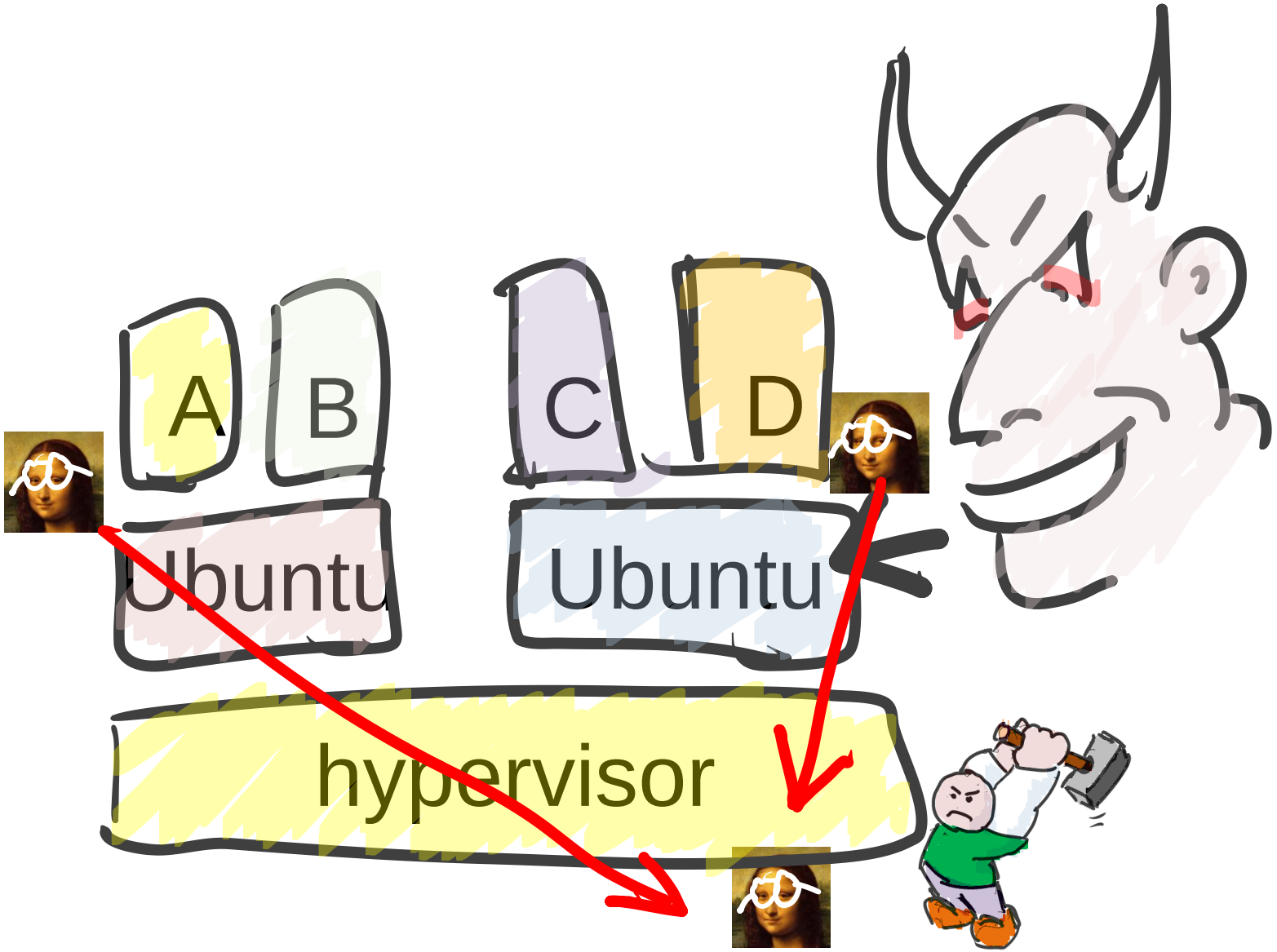
KVM / Clouds

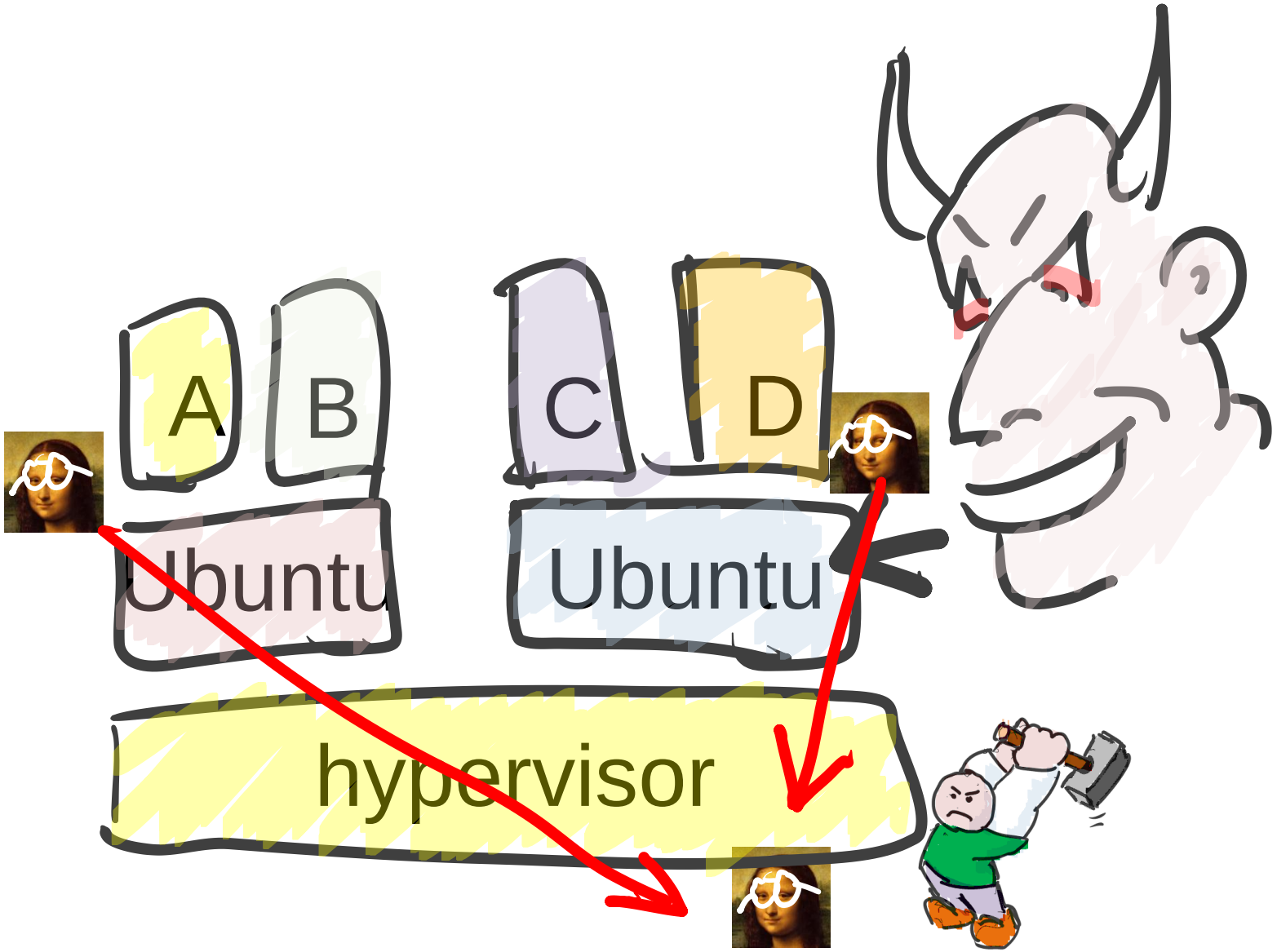
KSM: Kernel Same-page Merging




A
B
C
D
Ubuntu
Ubuntu
hypervisor

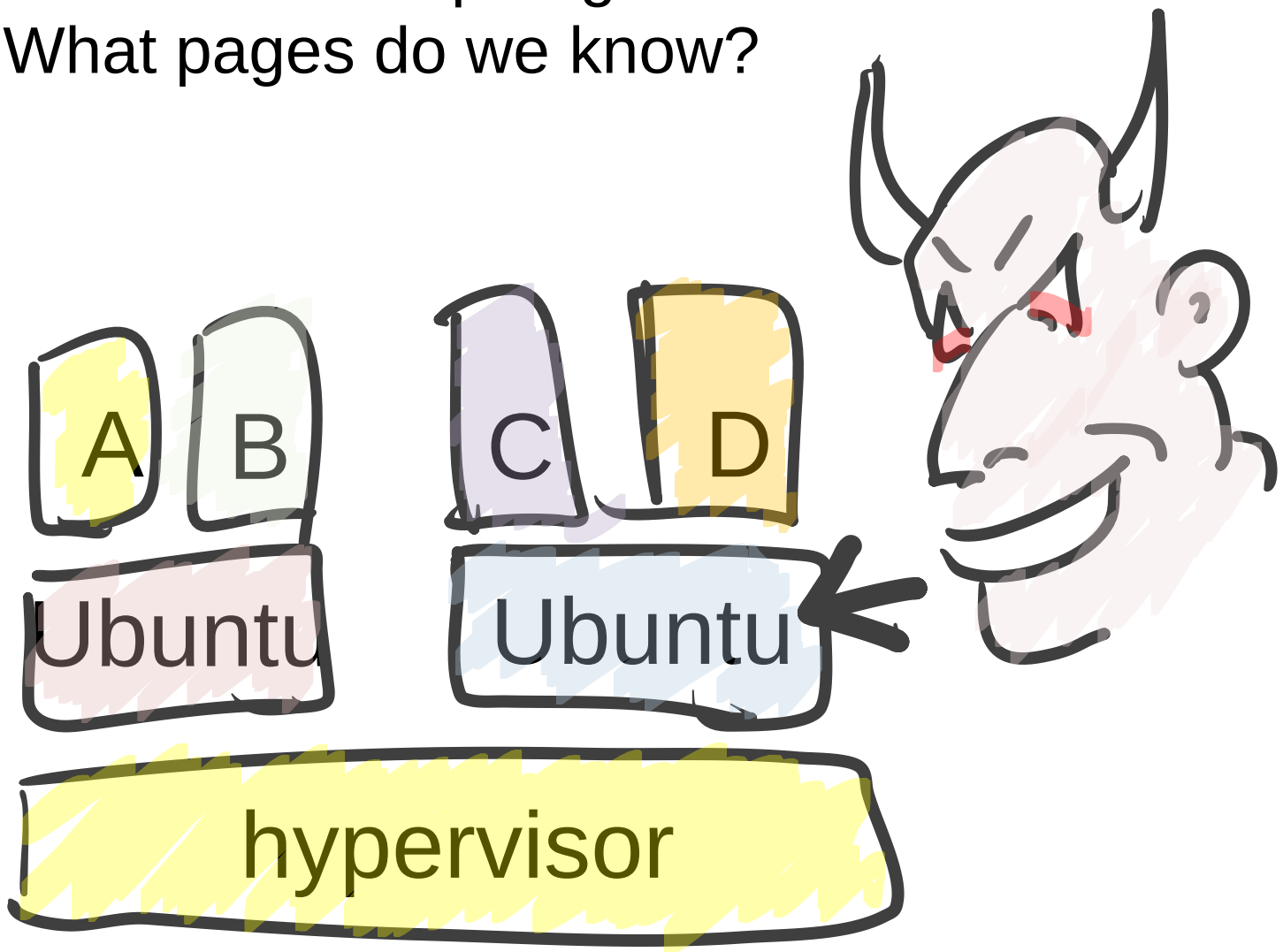

A
B
C
D
Ubuntu
Ubuntu
hypervisor

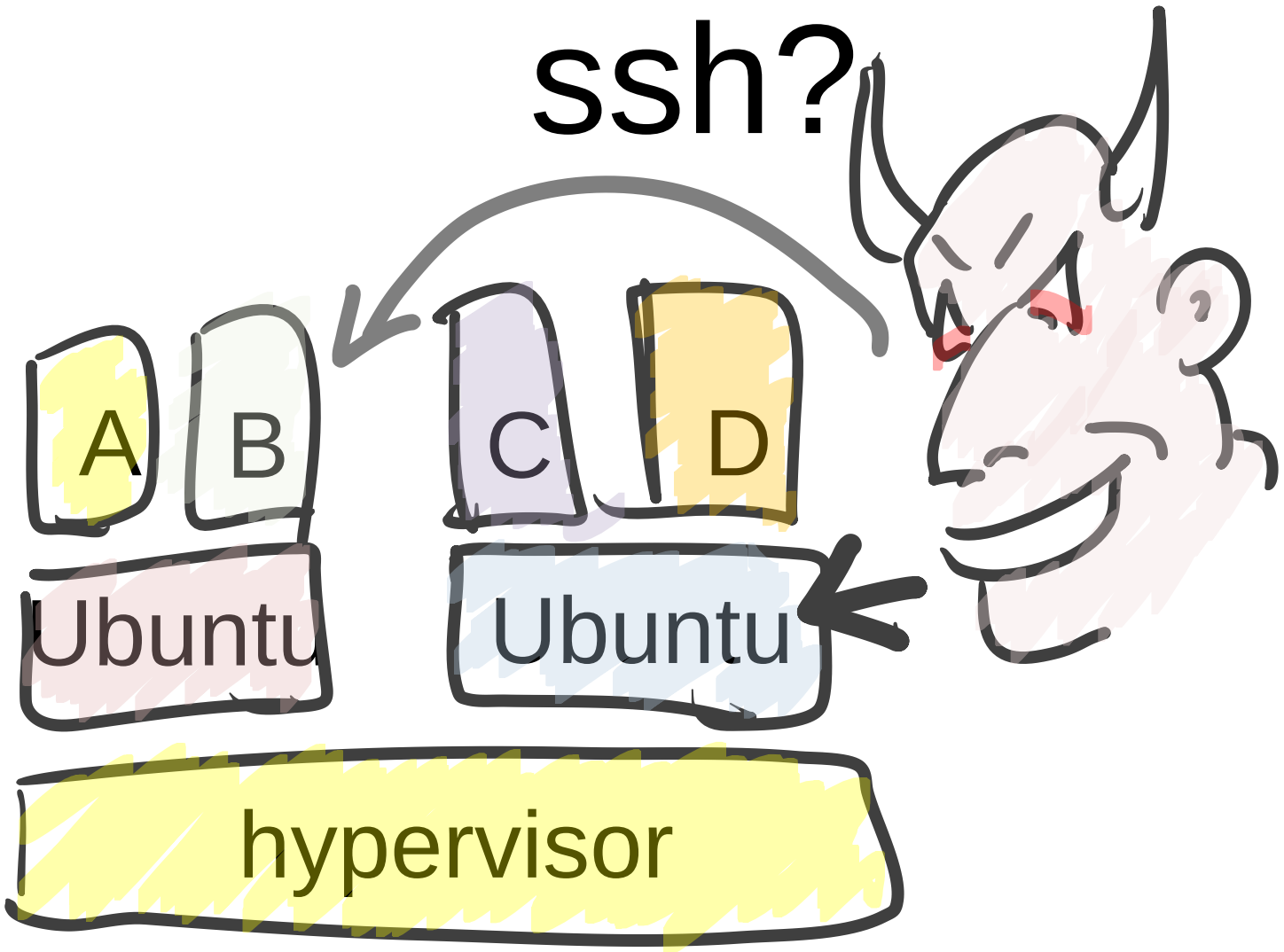

A
B
C
D
Ubuntu
Ubuntu
hypervisor

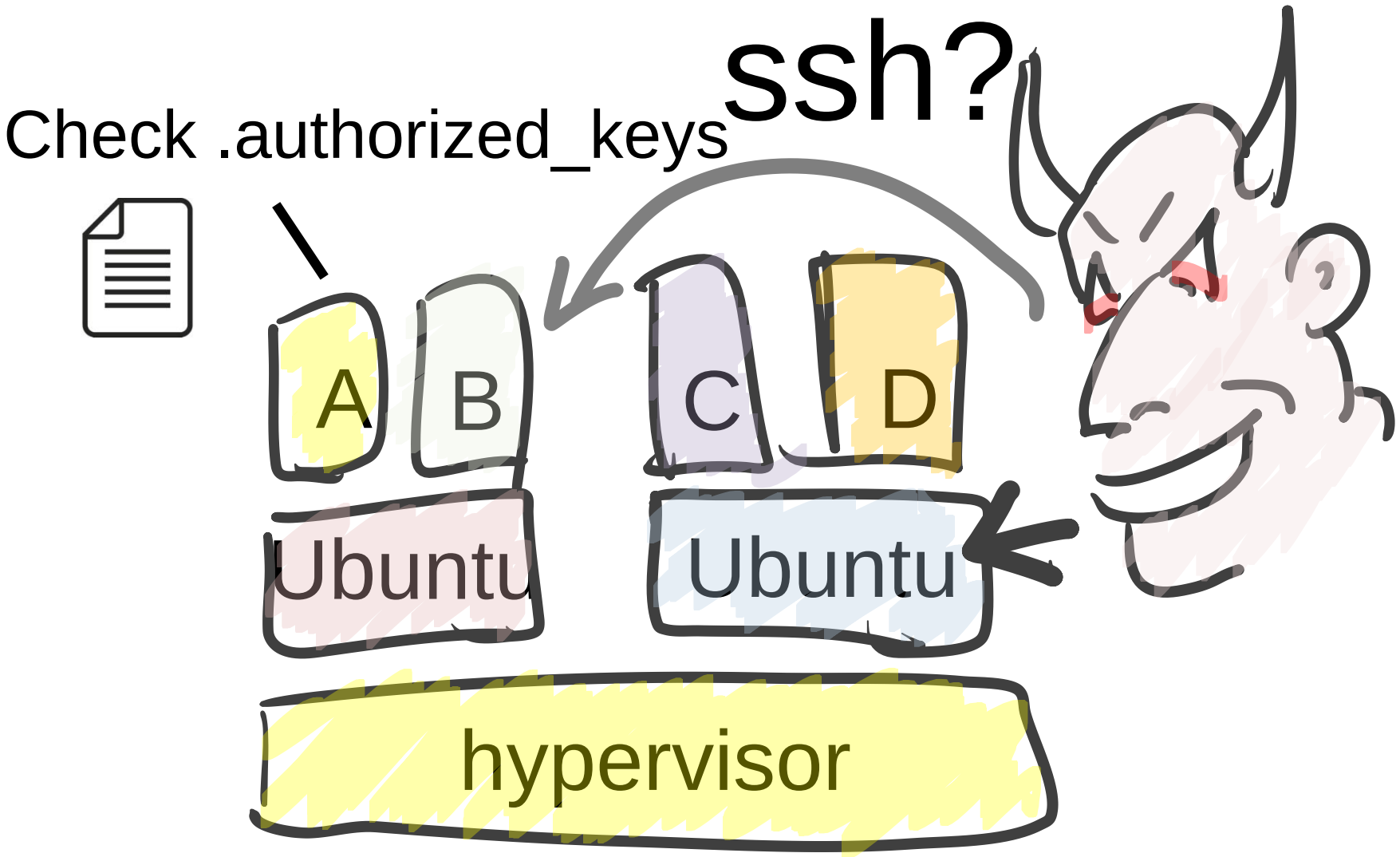

A
B
C
D
Ubuntu
Ubuntu
hypervisor

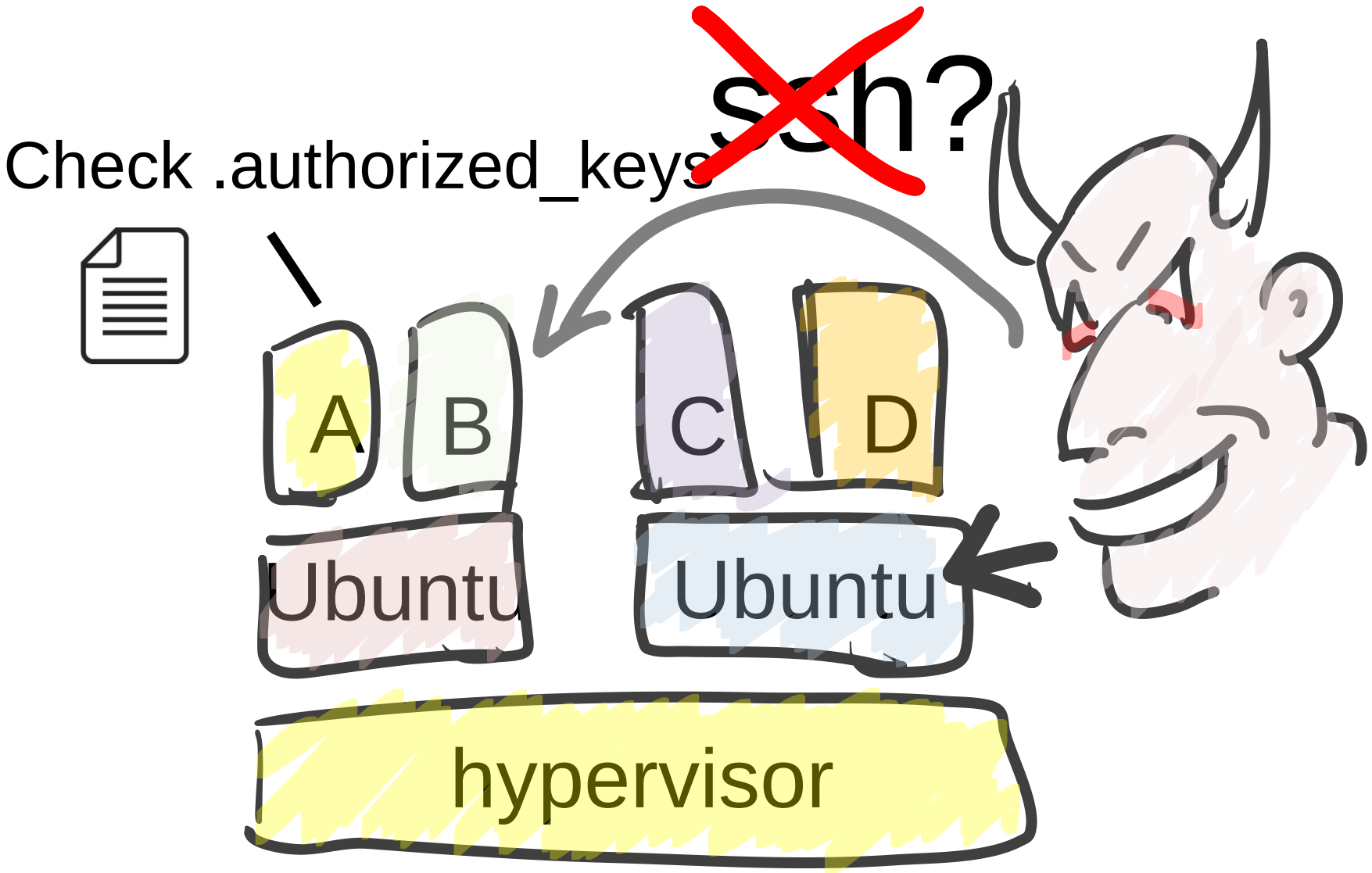
Questions:

1. What can we flip to gain access?
2. What pages do we know?




ssh?
A
B
C
D
Ubuntu
Ubuntu
hypervisor

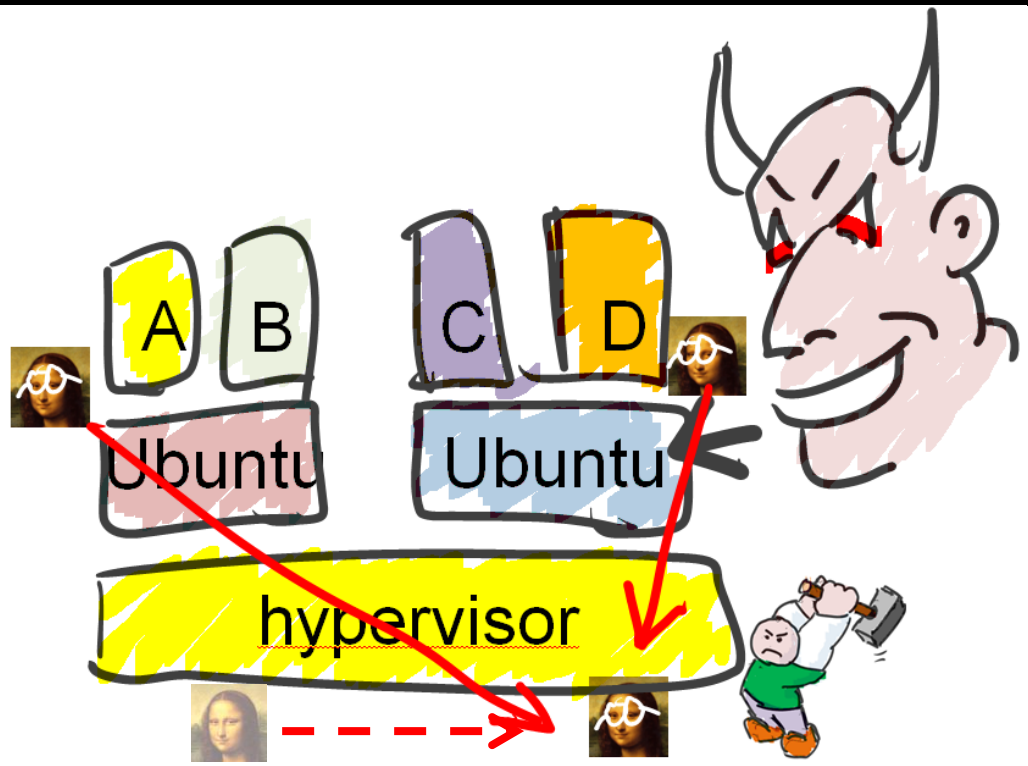

ssh?
Check .authorized_keys
A
B
C
D
Ubuntu
Ubuntu
hypervisor


Check .authorized_keys
ssh?
A
B
C
D
Ubuntu
Ubuntu
hypervisor

Now .authorized_keys is in memory

Public keys are not secret

Using dedup



We move it to a page susceptible to rowhammer

A bit flips in key…

Makes a **weak key**

Easy to generate private key

➔ We do this in minutes!

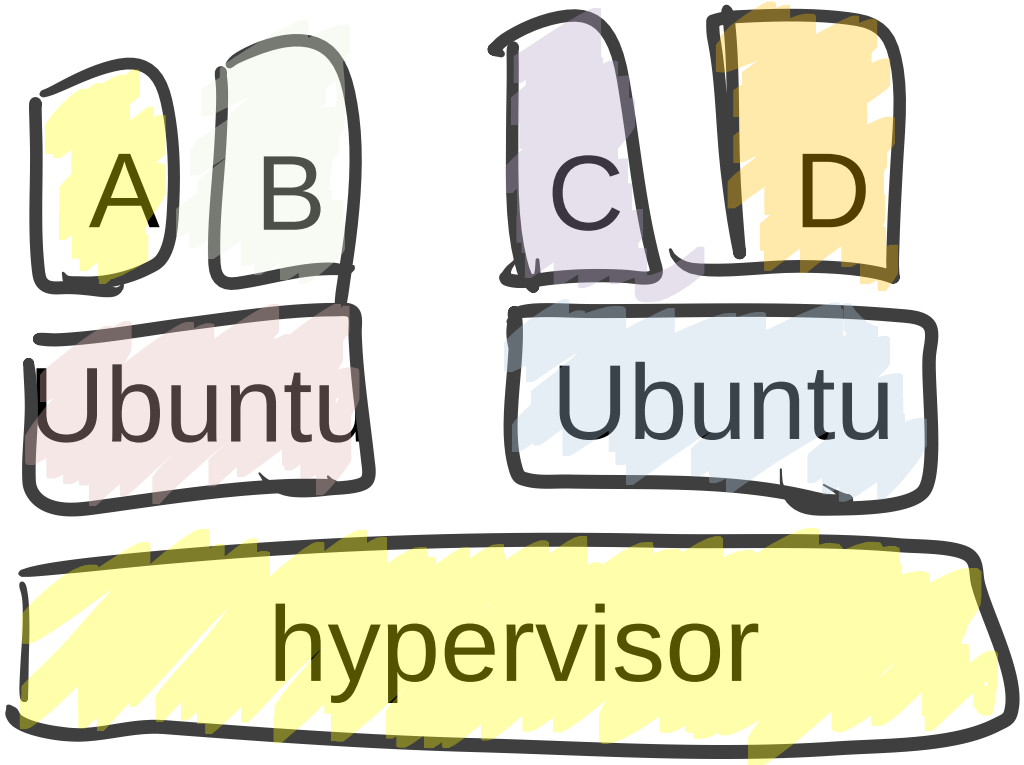
Better still…

debian.org

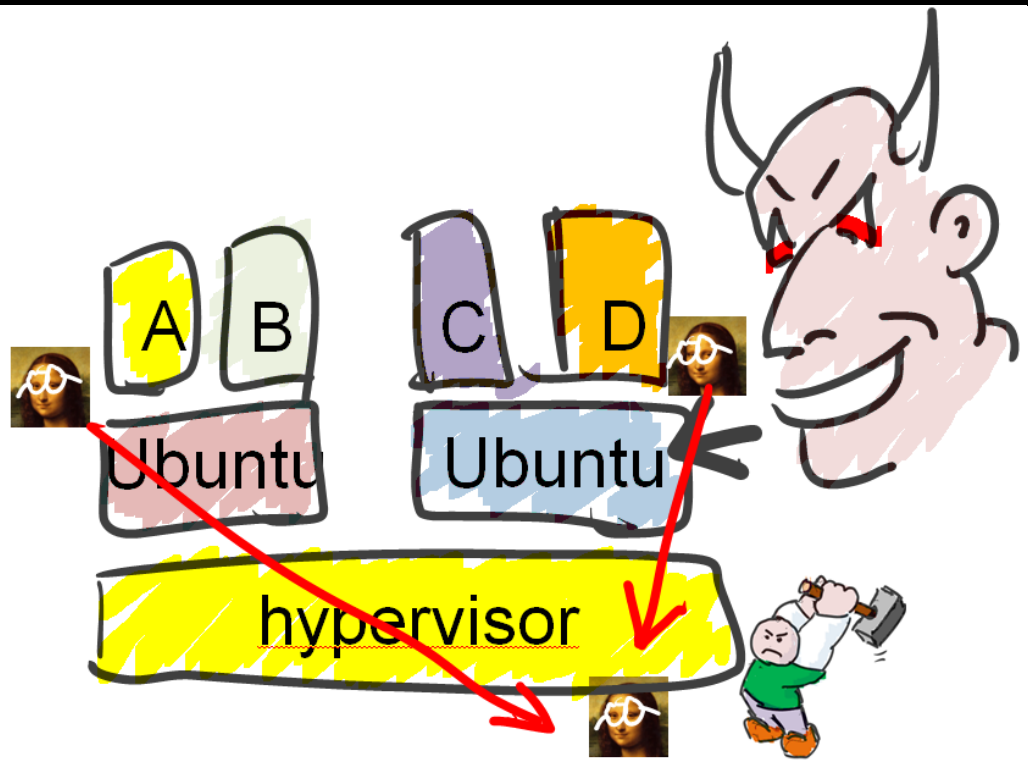
ubuntu.com

APT

`sources.list`: from which to install packages & update



Using dedup



We move `sources.list` to page susceptible to rowhammer

Hammer Time!

A bit flips…

Now we install from

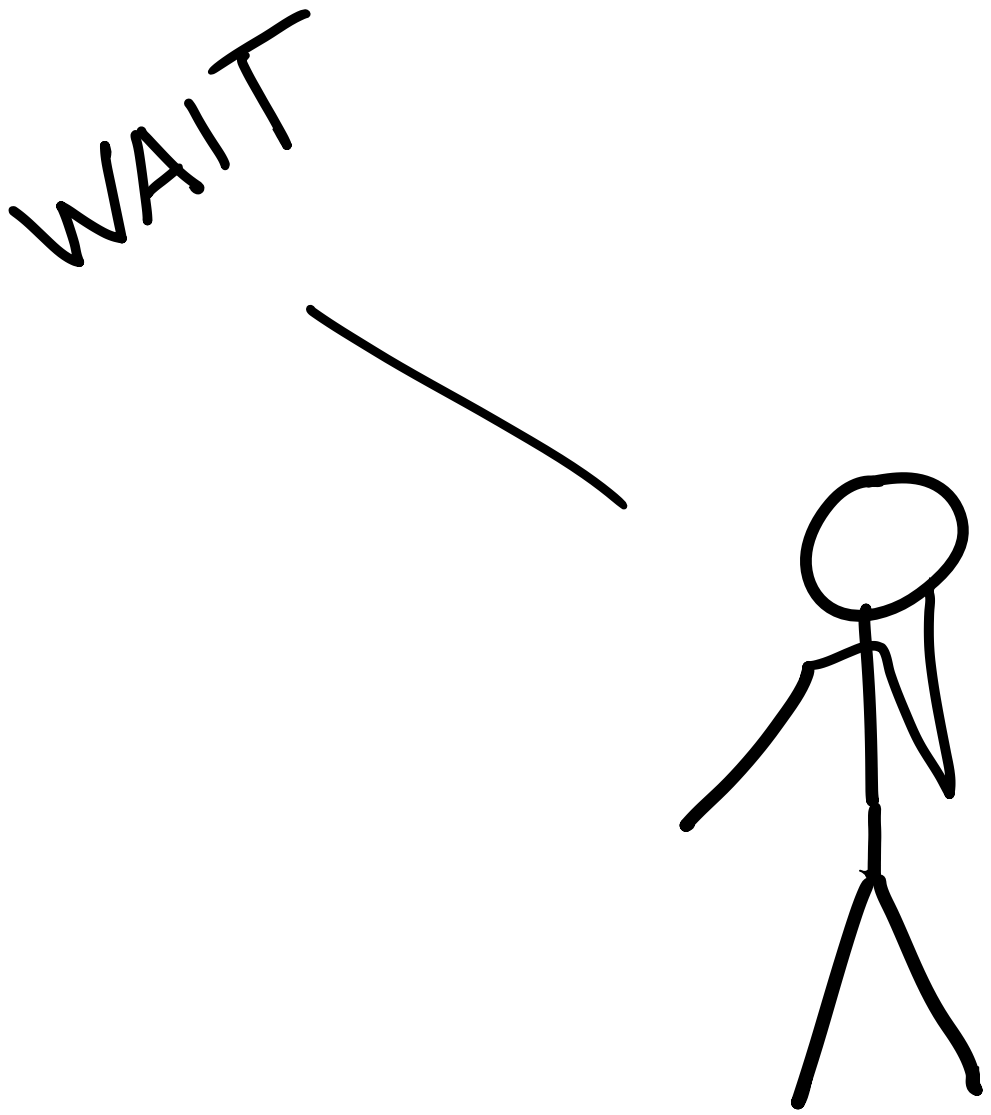
ubunvu.com

ucuntu.com

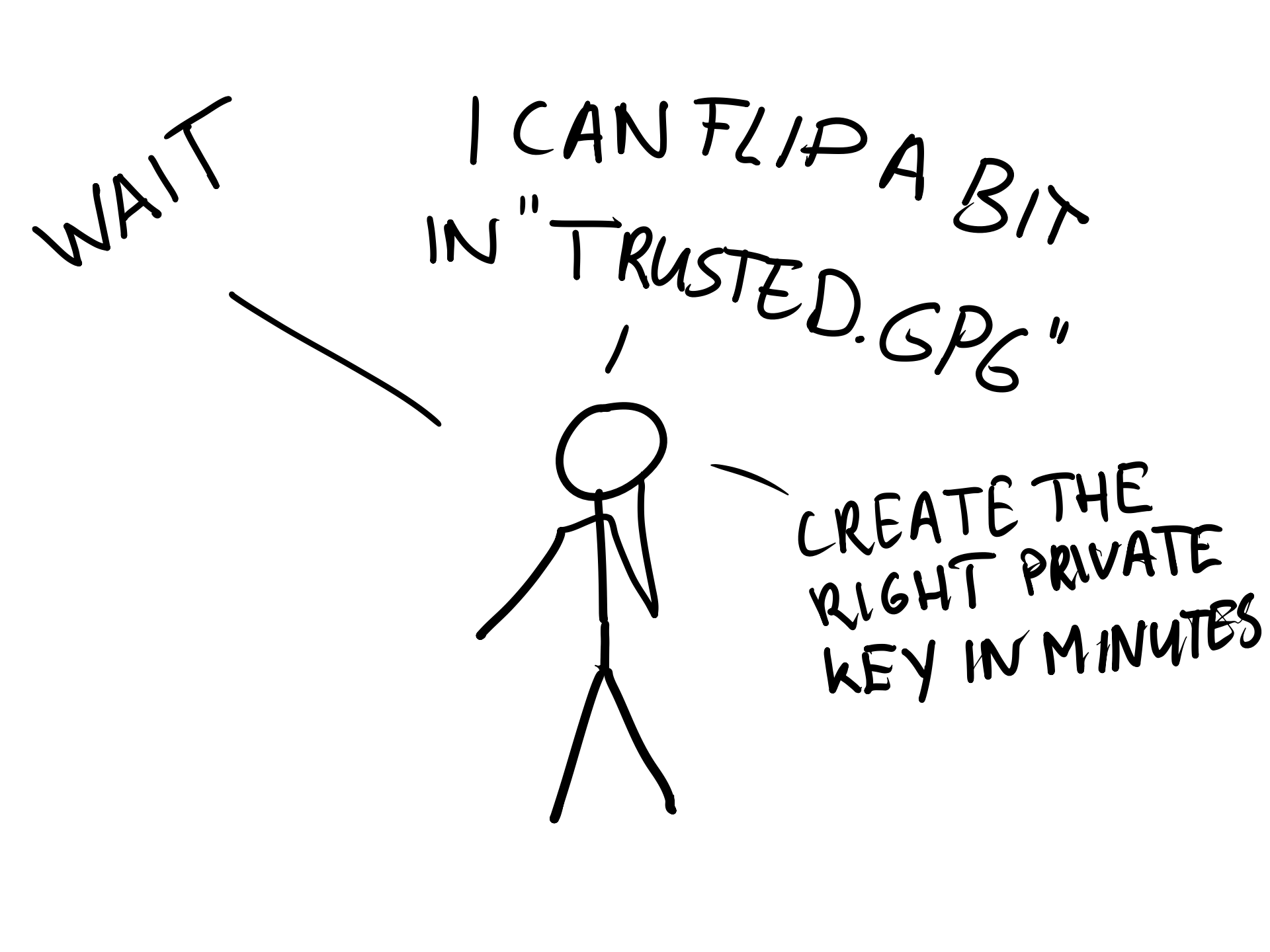
…

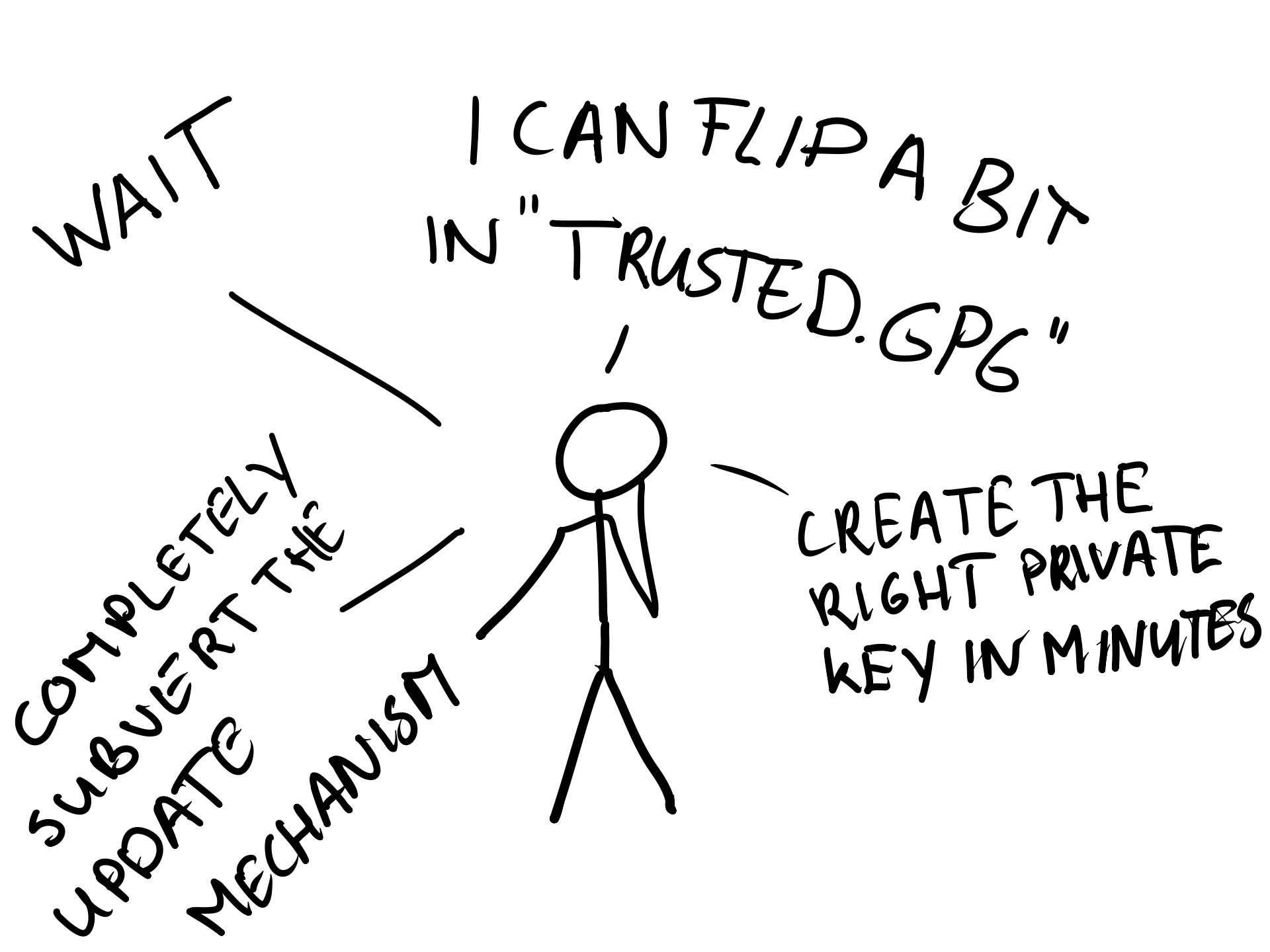
(which we own)

But fortunately, the packages are signed!


WAIT


WAIT
I CAN FLIP A BIT IN "TRUSTED.GPG"


WAIT
I CAN FLIP A BIT IN "TRUSTED.GPG"
CREATE THE RIGHT PRIVATE KEY IN MINUTES


WAIT
I CAN FLIP A BIT IN "TRUSTED.GPG"
COMPLETELY SUBVERT THE UPDATE MECHANISM
CREATE THE RIGHT PRIVATE KEY IN MINUTES

Flip Feng Shui: Impact

Notified:

Red Hat, Oracle, Xen, VMware, Debian, Ubuntu, OpenSSH, GnuPG, hosting companies

NCSC did all the hard work, thanks!

GnuPG "*included hw bit flips in their threat model*"

gpgv: Tweak default options for extra security.

```
author    NIIBE Yutaka <gniibe@fsij.org>
          Fri, 8 Jul 2016 20:20:02 -0500 (10:20 +0900)
committer NIIBE Yutaka <gniibe@fsij.org>
          Fri, 8 Jul 2016 20:20:02 -0500 (10:20 +0900)
commit    e32c575e0f3704e7563048eea6d26844bdfc494b
```

Mitigations

“Can we just disable memory deduplication and buy better DRAM?”

Yes, you really should, but...

Mitigations

No dedup?

Need another memory massaging primitive

Check our DRAMMER paper ;)

CCS 2016

Mitigations

Better DRAM?

- Not so fast
  - Rowhammer exploits fundamental DRAM properties
- Discovered on DDR3, still there on DDR4
  - Despite targeted countermeasures
- Originally on x86, we found flips on ARM

Mitigations

No dedup and no Rowhammer?

Other primitives will come along

Expect:

More hw/sw properties you didn't know about

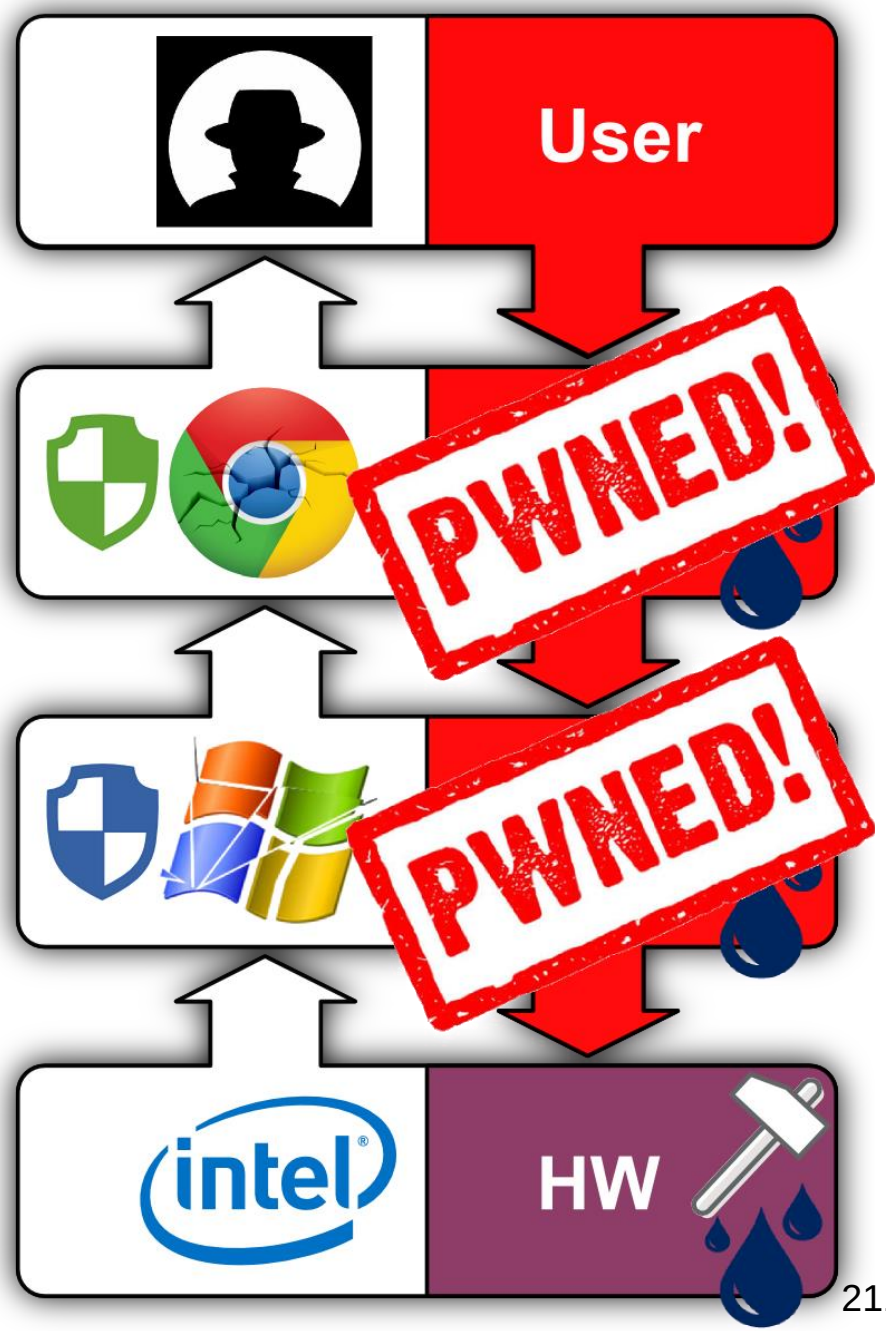
More **side channels**

More **hardware glitches**

A **radical change** in the way we think about sys security and "reasonable" threat models

Flip Feng Shui:

Is Physics Part of Your Threat Model Yet?



Rethinking Systems Security

Software security defenses

[Aug 4, 12:00] **Microsoft**: "*Thanks to our mitigation improvements, since releasing Edge one year ago, there have been no zero day exploits targeting Edge*"

Rethinking Systems Security

Software security defenses

[Aug 4, 12:00] **Microsoft**: "*Thanks to our mitigation improvements, since releasing Edge one year ago, there have been no zero day exploits targeting Edge*"

[Aug 4, 17:00] **VUSec**: "*Dedup Est Machina: One can exploit the latest Microsoft Edge with all the defenses up, even in absence of software/configuration bugs*"

Rethinking Systems Security

Formally verified systems

Rethinking Systems Security

Formally verified systems

[Aug 10] **VUSec**: "*Flip Feng Shui: Reliable exploitation of bug-free software systems*"

Conclusion

Software security defenses are getting better

- But hw and sw are getting extremely complex
- Potentially huge unexplored attack surface
- Attackers can subvert even “perfect” software
- Beyond side channels (but they play a role)

https://vusec.net