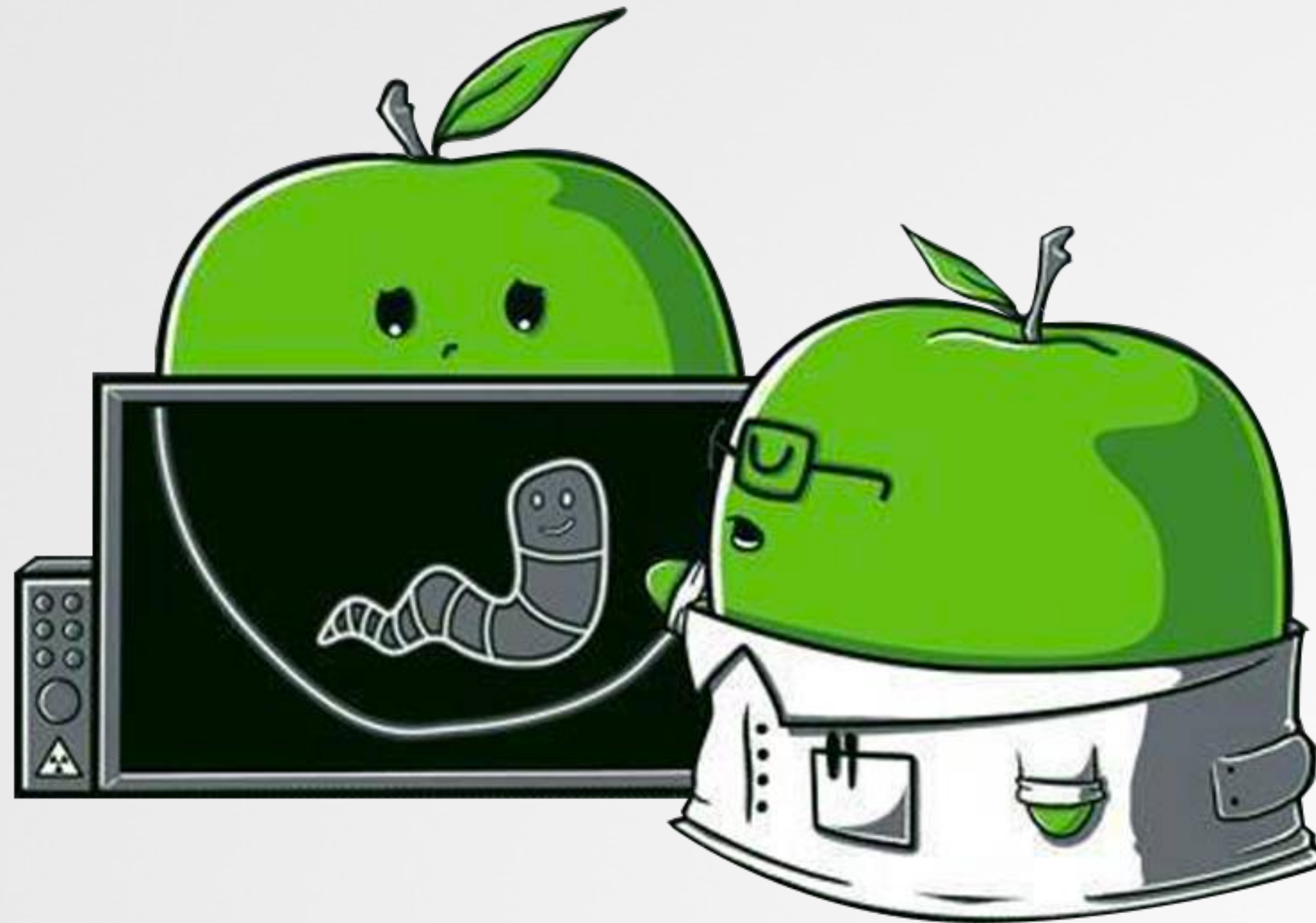


Arm'd & Dangerous

analyzing arm64 malware targeting macOS



WHOIS



 @patrickwardle



Objective-See

tools, blog, & malware collection



#OBTS

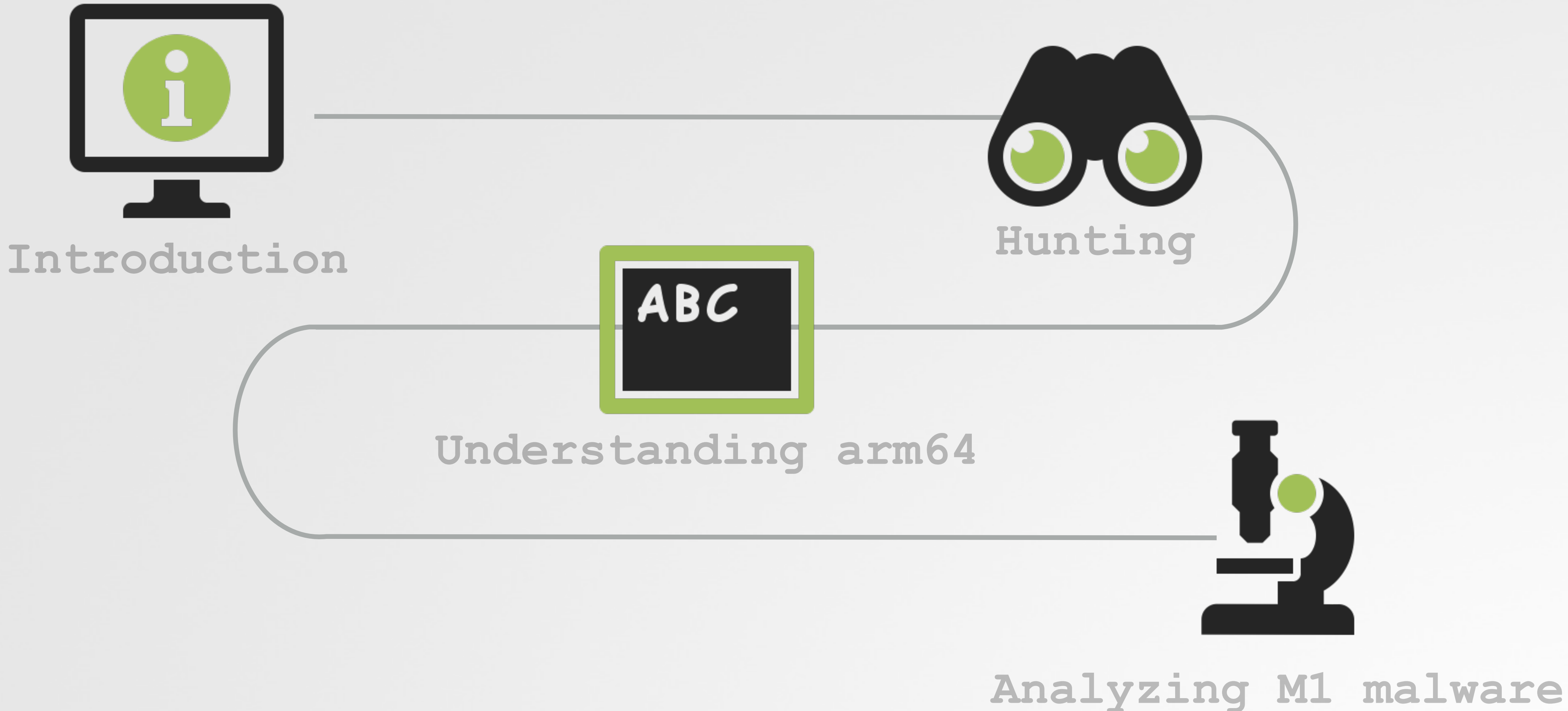
"Objective by the Sea"
(macOS security conference)



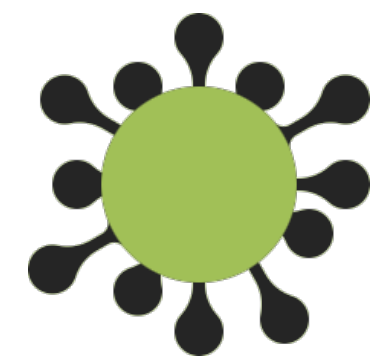
Book(s) :

"The Art of Mac Malware"

OUTLINE

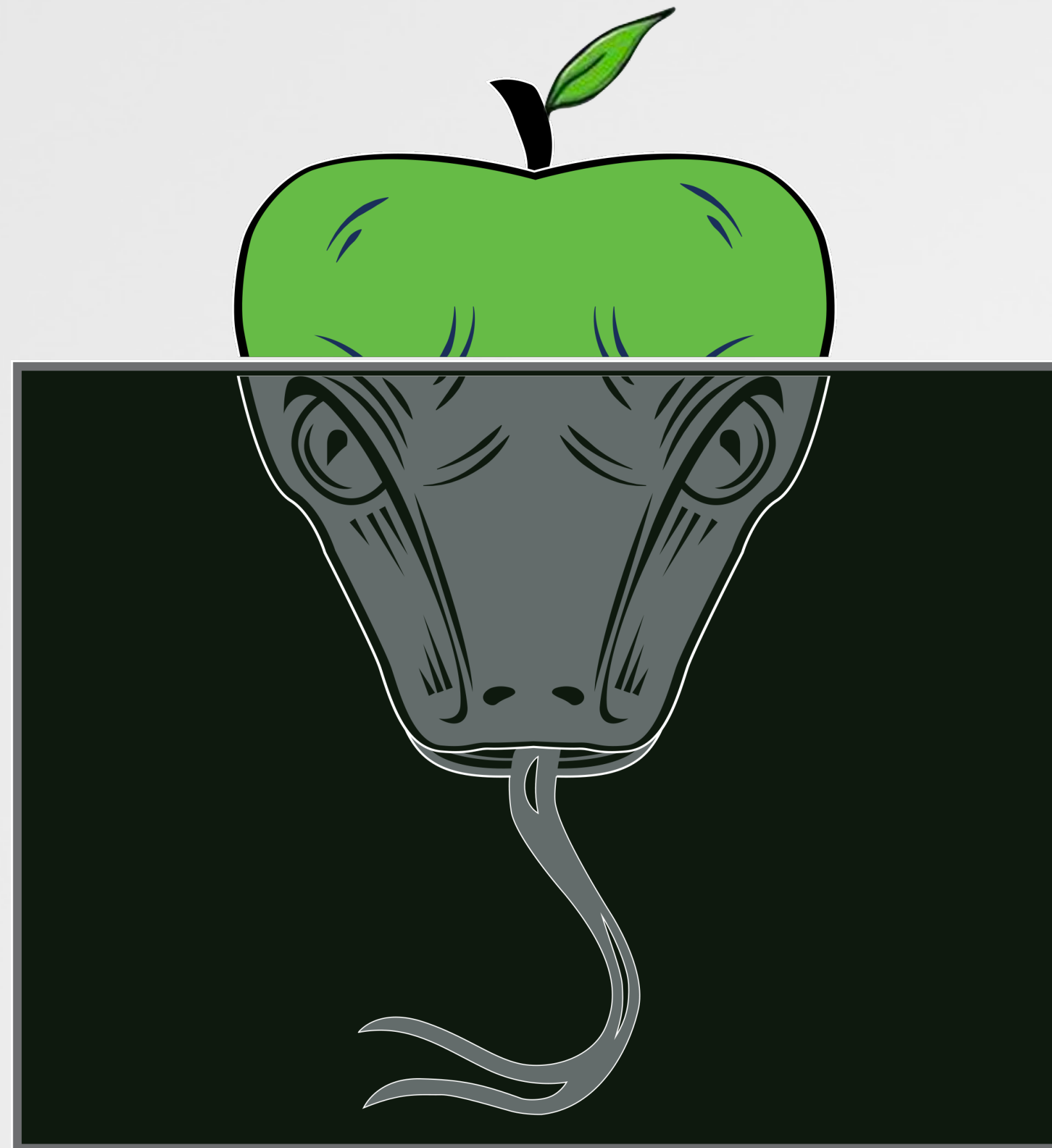


"M1 malware" defined:



Malware natively compiled to run on Apple Silicon.
More specifically, arm64 malware targeting Macs (macOS).

Introduction

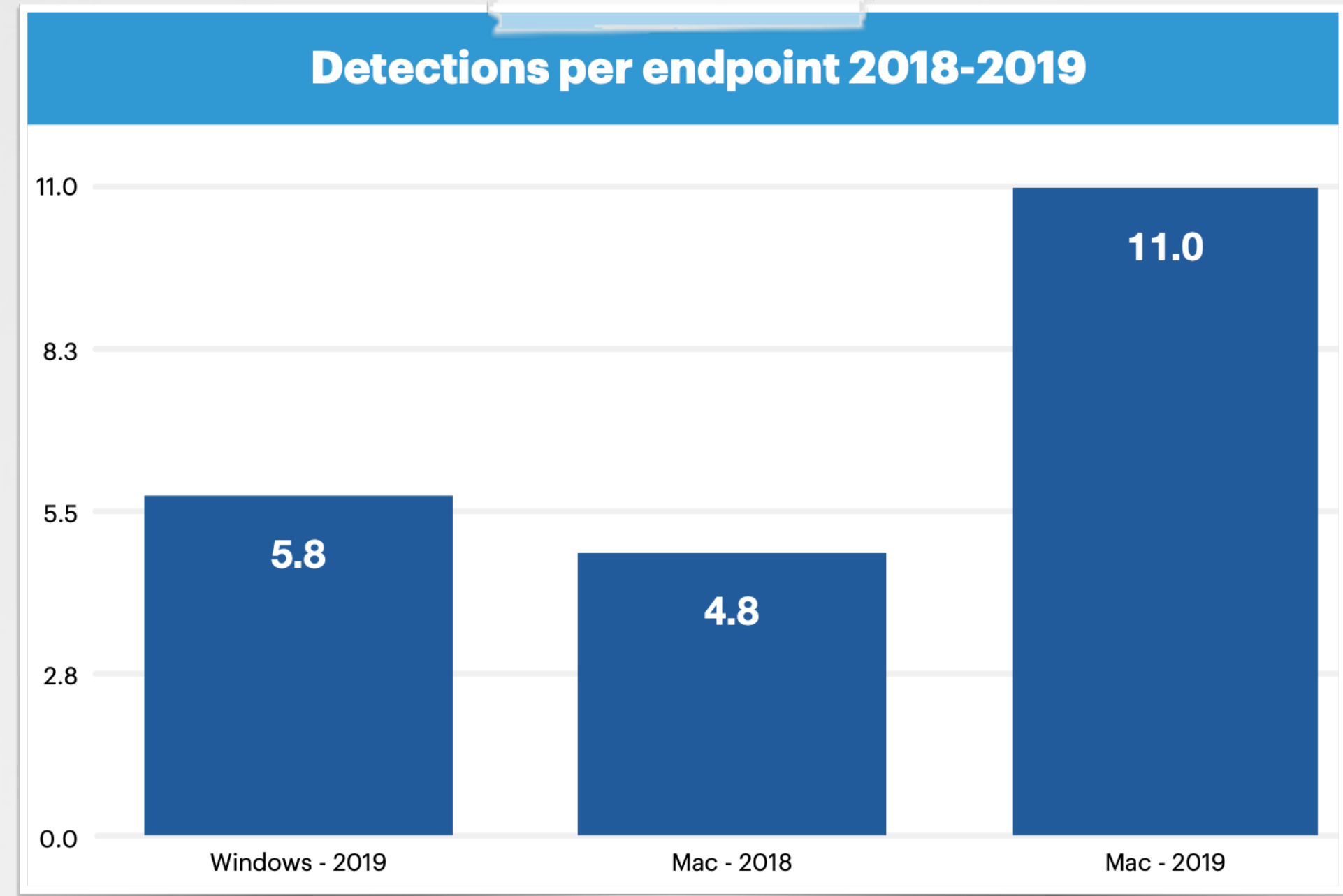


THE GROWTH OF MACS

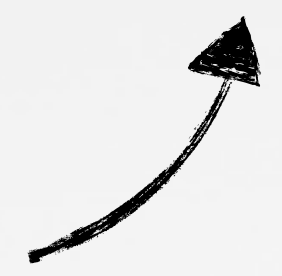
...and unsurprisingly, the growth of macOS malware

9T@5Mac ✓
IDC: Mac shipments rise industry-leading 49% in Q4

More Macs



more than Windows!?



More Mac Malware
(credit: MalwareBytes)



As macOS becomes more prevalent, (rather obviously), so too does malware targeting this platform.

THE GROWTH OF MACS ...driven largely by [the] M1?

Mac sales are surging thanks to Apple's new M1 processor

Mac Sales Skyrocketing After M1 Launch

Apple's worldwide Mac shipments grew massively in the fourth quarter of 2020 after the launch of three new Macs with the [M1](#) chip, according to new PC shipping estimates [shared by Gartner](#). Apple shipped an estimated 6.9 million Macs, up from the 5.25 million it shipped at the same time in 2019, marking significant growth of 31.3 percent.

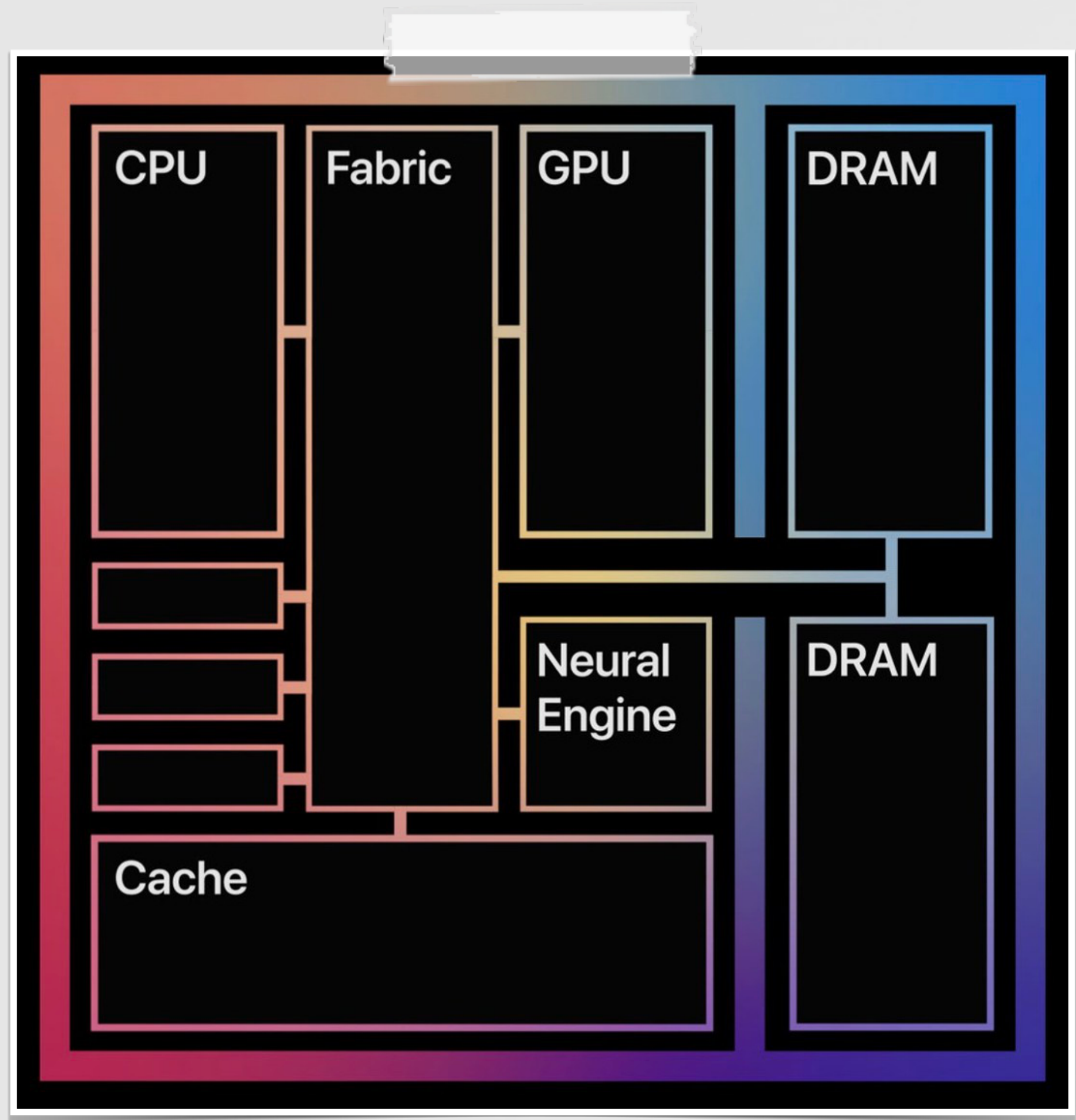
The Mac is back — and the M1 chip gets the credit



"Fueled by the M1, we set an all-time [Mac] revenue record continuing the momentum for the product category" -Tim Cook (Apple)

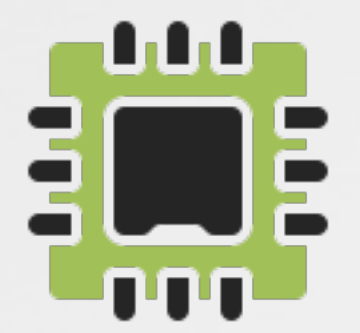
WHAT IS M1? (AKA "APPLE SILICON")

an arm-based system on a chip (SoC)



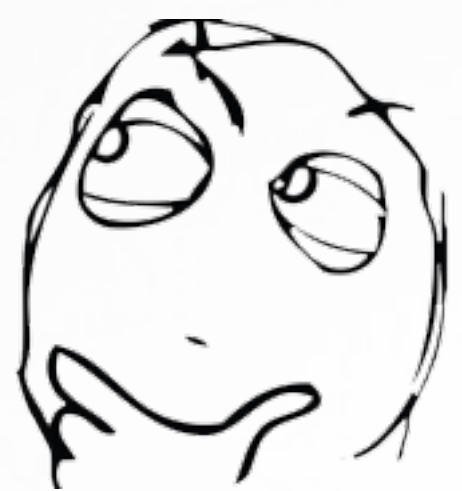
The M1 chip

Multiple technologies combined on a single chip



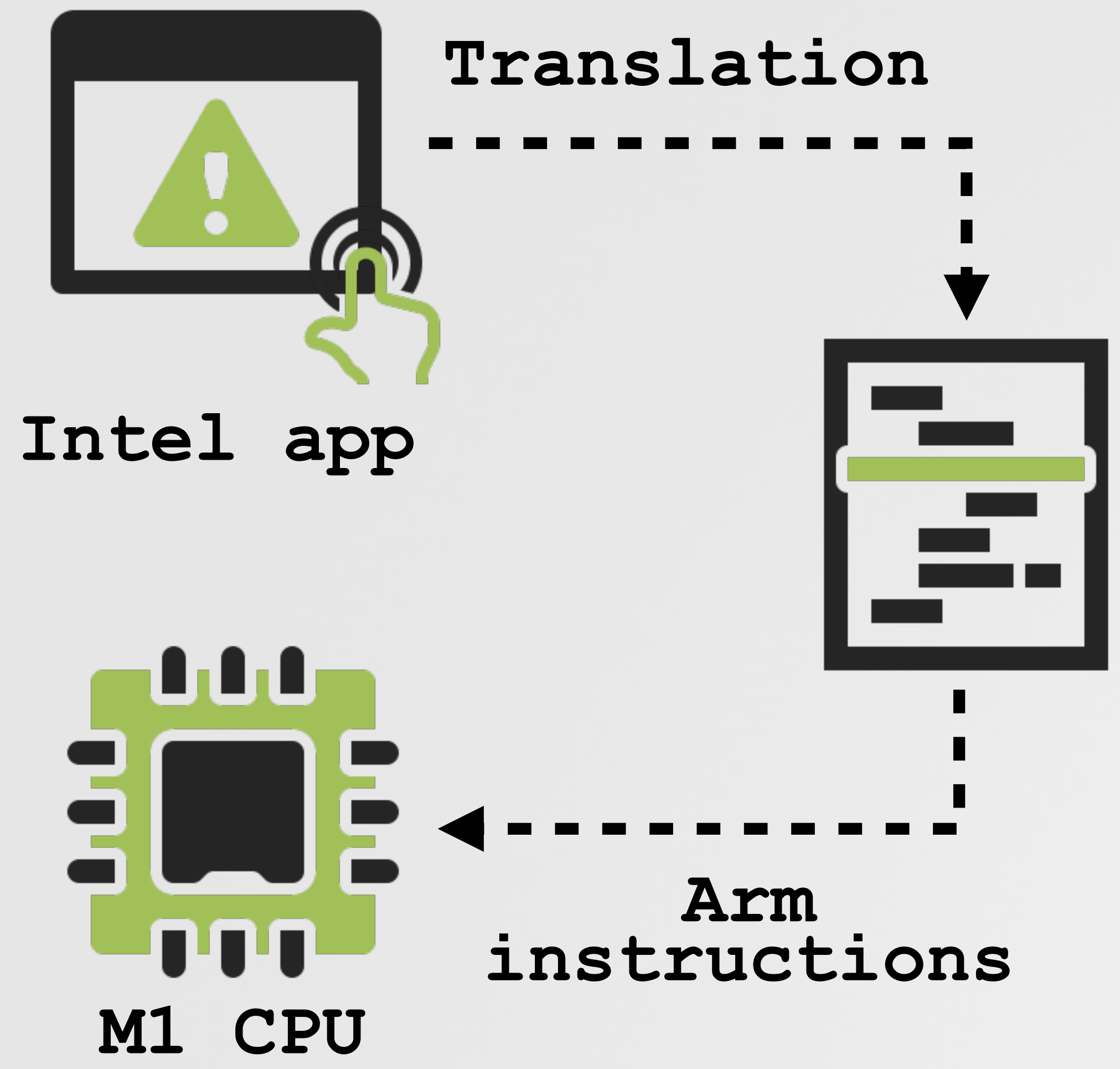
CPU:
arm64 instruction set

...malware native to this CPU, will disassemble into Arm (vs. Intel).



AND ROSETTA (2)

...run intel-based apps on apple silicon



Translation -> slow-down



(was) Prone to crashes

```
Process: oahd-helper [36752]
Path: /Library/Apple/*/oahd-helper
Identifier: oahd-helper
Version: 203.13.2
Code Type: ARM-64 (Native)
Parent Process: oahd [506]
Responsible: oahd [506]
User ID: 441

Date/Time: 2021-02-12 10:34:15.107 -1000
OS Version: macOS 11.1 (20C69)

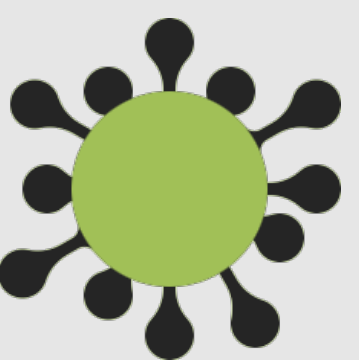
Crashed Thread: 0 Dispatch queue: com.apple.main-thread
```

Rosetta (oahd-helper) crash

 "not a substitute for creating a native version of your app" -apple

WHY TALK ABOUT M1 MALWARE

well, several important reasons!



- Is (was) inevitable
- no rosetta crashes
- native code, faster

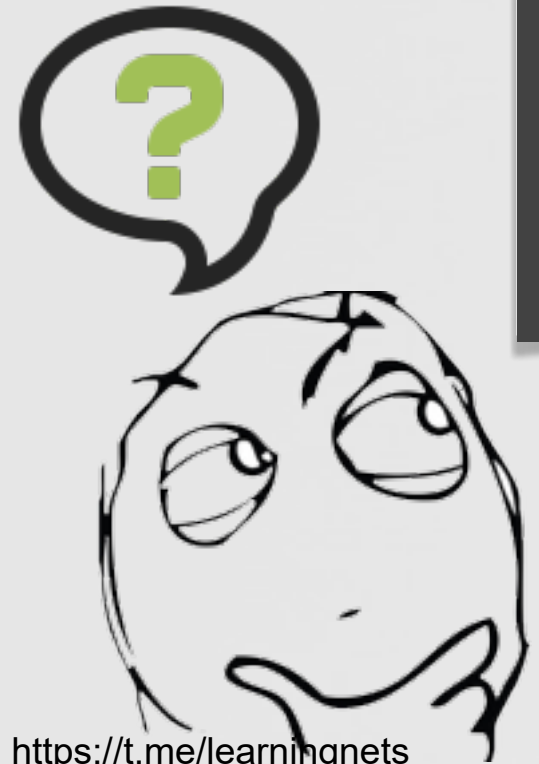


Disassembly is arm64

```

01  movz    x0, #0x1a
02  movz    x1, #0x1f
03  movz    x2, #0x0
04  movz    x3, #0x0
05  movz    x16, #0x0
06  movz    x16, #0x0
07  movz    x16, #0x0
08  svc     #0x80

```



an unfamiliar instructions set!?



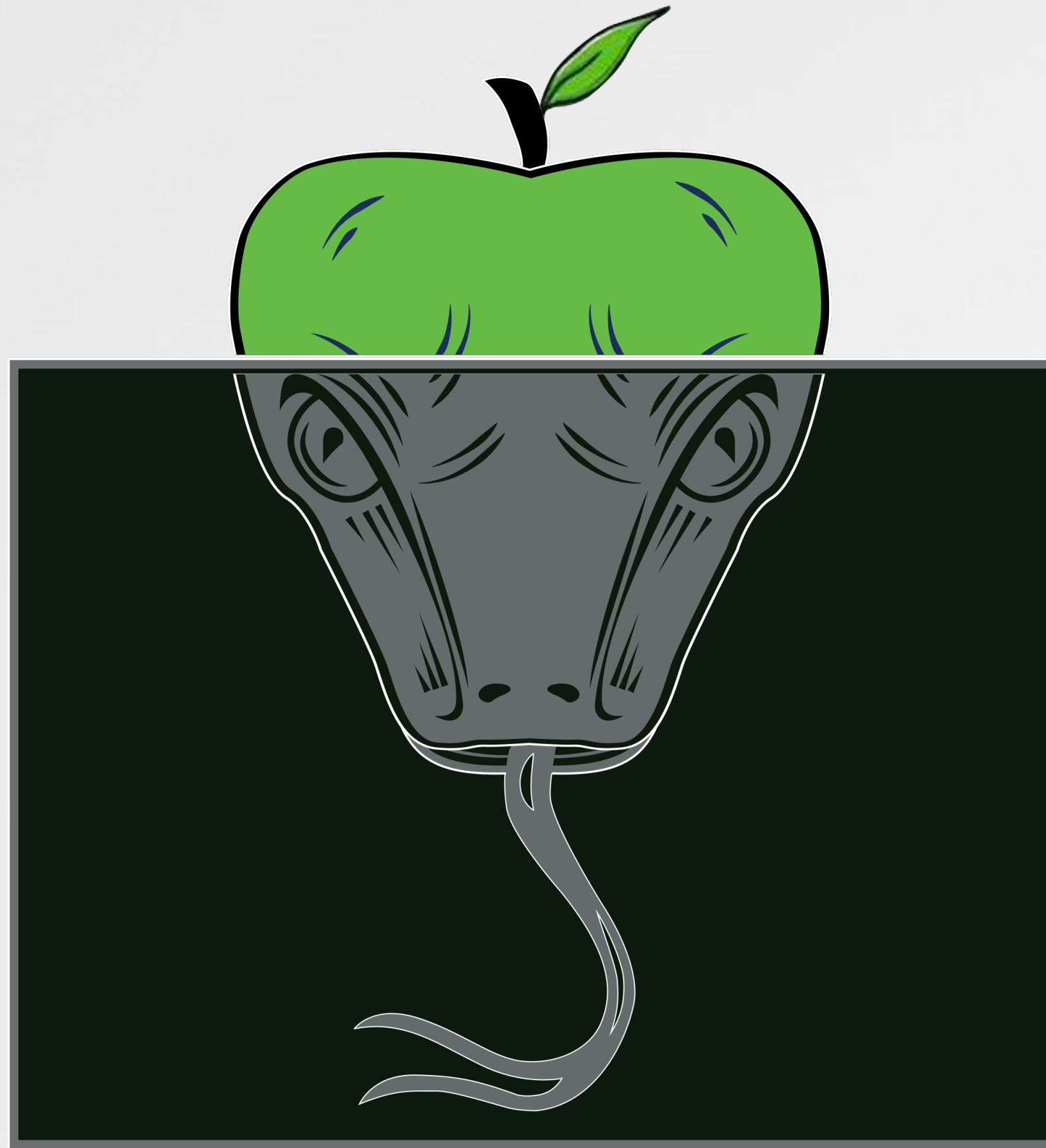
Missed AV detections (known malware: 10%+ drop)

Intel version: detected

Arm version: not detected

} same malware

Hunting for M1 malware



HOW TO IDENTIFY M1 CODE

in short, a macOS binary with arm64/e

```
% file Calculator.app/Contents/MacOS/Calculator
Mach-O universal binary with 2 architectures:
Mach-O 64-bit executable x86_64
Mach-O 64-bit executable arm64e

% lipo -archs Calculator.app/Contents/MacOS/Calculator
x86_64 arm64e
```

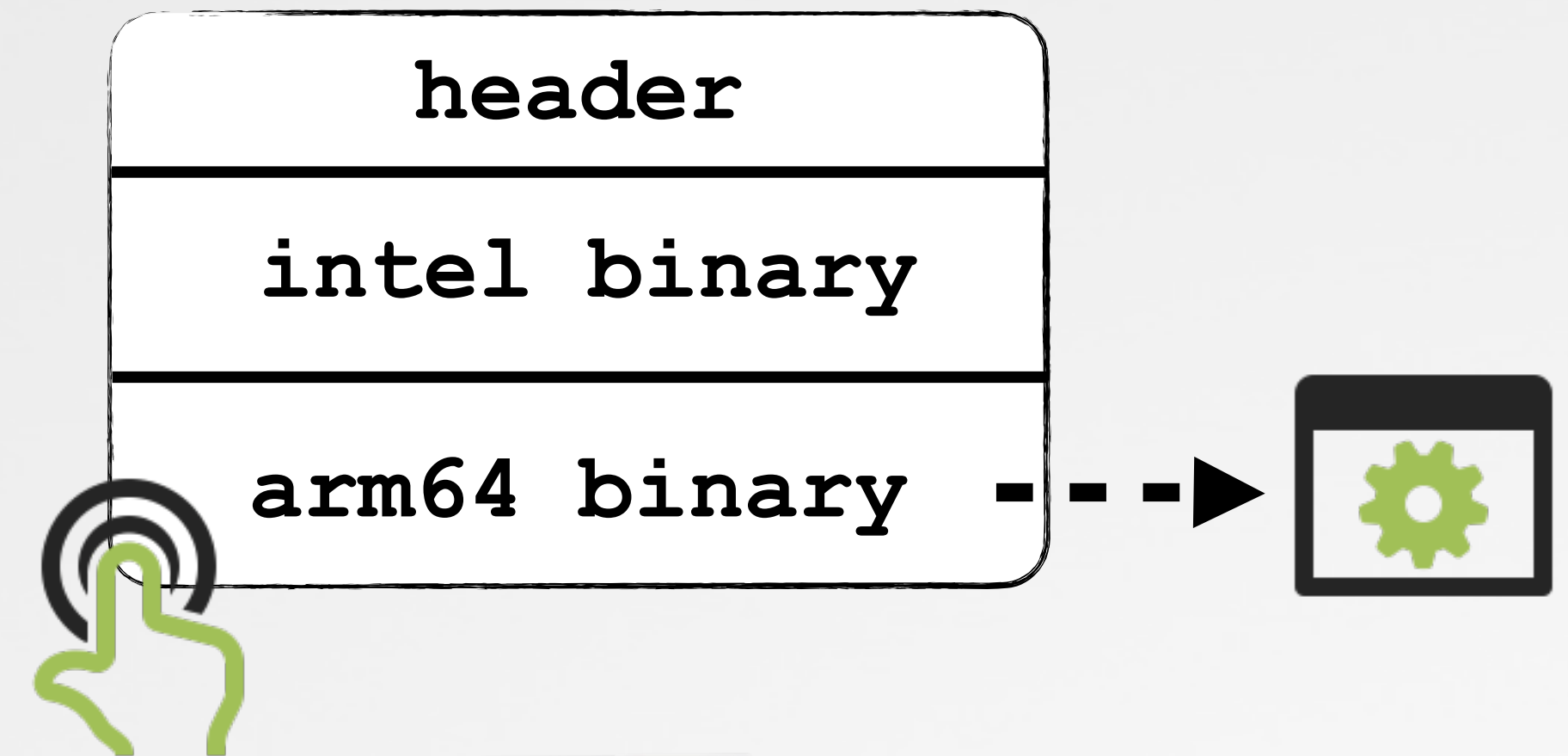
arm64/arm64e code
(may be found in
universal binary)

What's arm64e?



arm64 enhanced
+pointer auth, etc.

Universal binary



```
% otool -lv Calculator.app/Contents/MacOS/Calculator
...
Load command 10
    cmd LC_BUILD_VERSION
    cmdsize 32
    platform MACOS
    minos 11.4
    sdk 11.4
```

...built for macOS
(also: LC_VERSION_MIN_MACOSX)

HUNTING FOR M1 MALWARE

querying virustotal for specimens



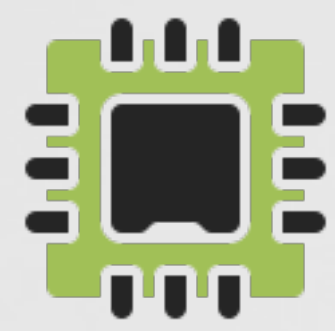
type:macho tag:arm tag:64bits tag:multi-arch NOT engines:IOS positives:2+



tag:macho
apple executable



tag: multi-arch
universal binary



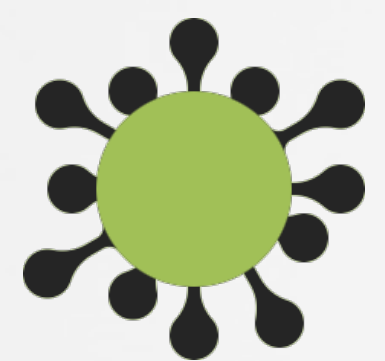
tag:arm
contains arm code



NOT engines:IOS
not an iOS binary



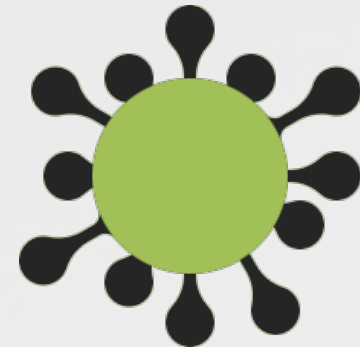
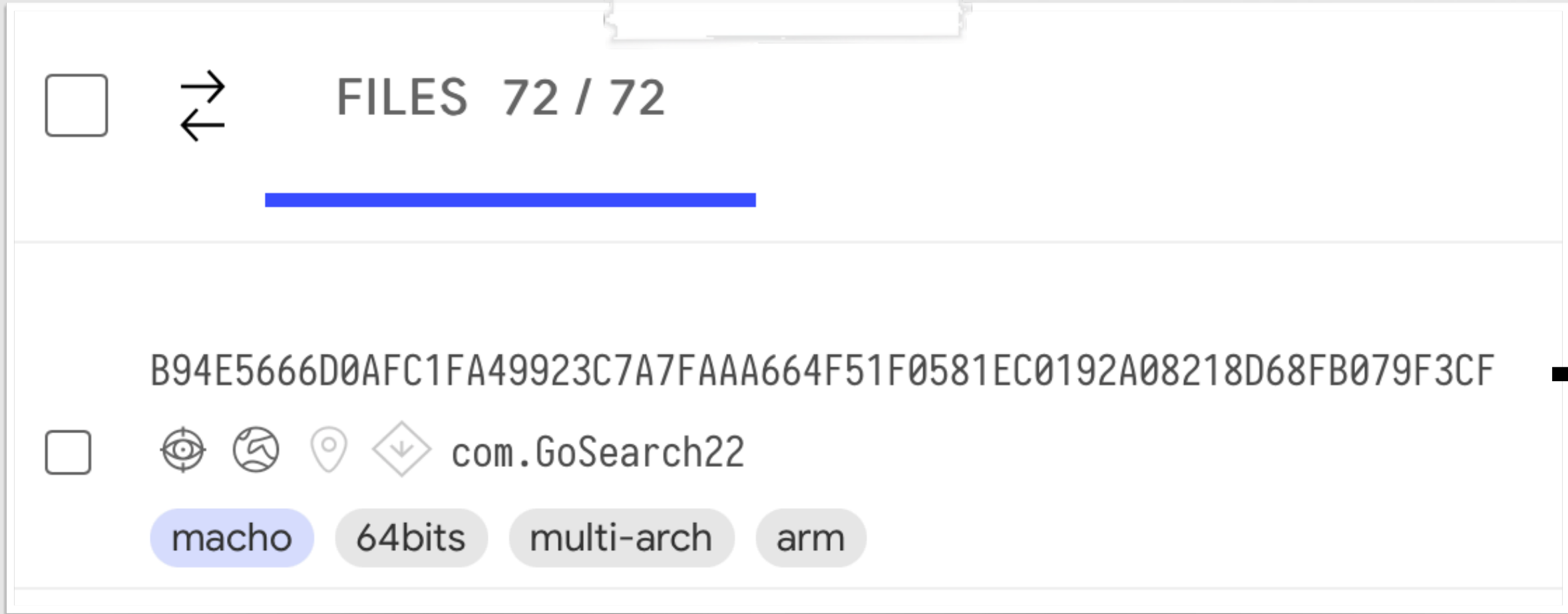
tag:64bits
contains 64bit code



positives:2+
flagged by 2+ AV engines

TRIAGING GOSEARCH22

a candidate (M1) binary



Flagged by several AV engines (intel code?)
+ app's cert. revoked -

```
% file GoSearch22
Mach-O universal binary with 2 architectures:
[arm64:Mach-O 64-bit executable arm64]
[x86_64:Mach-O 64-bit executable x86_64]

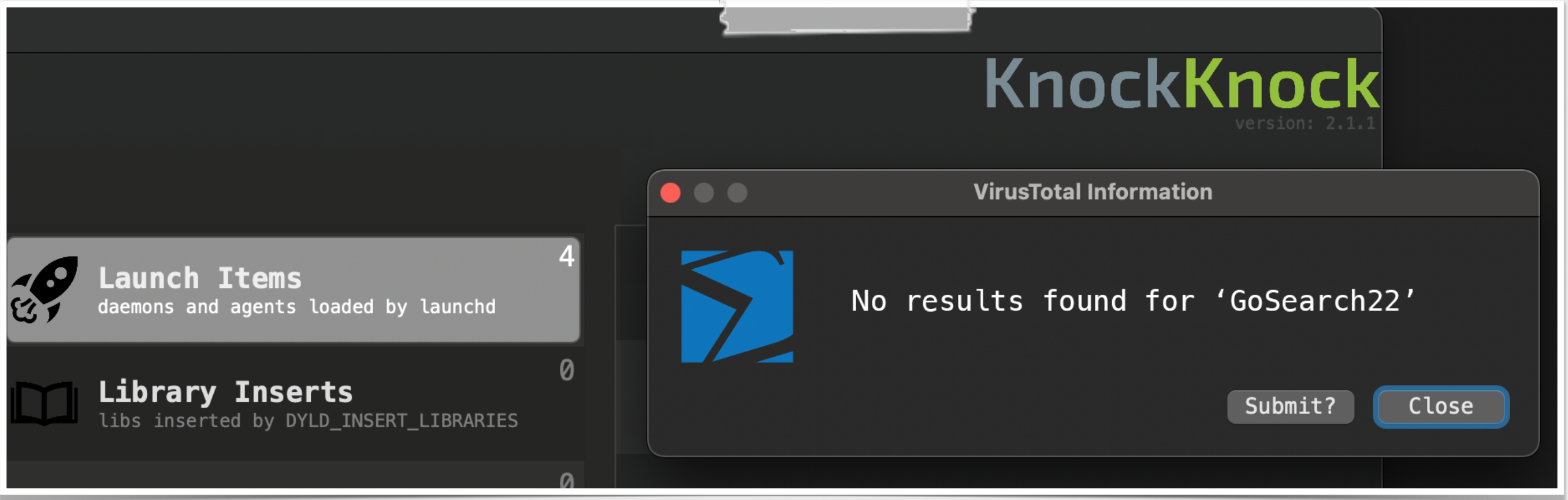
% otool -lv GoSearch22
...
Load command 9
  cmd LC_VERSION_MIN_MACOSX
  version 10.12
```



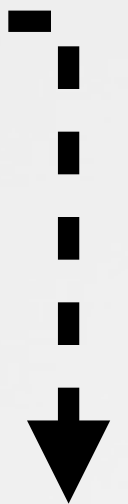
Certificate revoked (by Apple)

Universal macOS binary (with arm64)

HOW DID IT END UP ON VIRUSTOTAL? ...detected and submitted via KnockKnock!



KnockKnock detection
(free: objective-see.com)



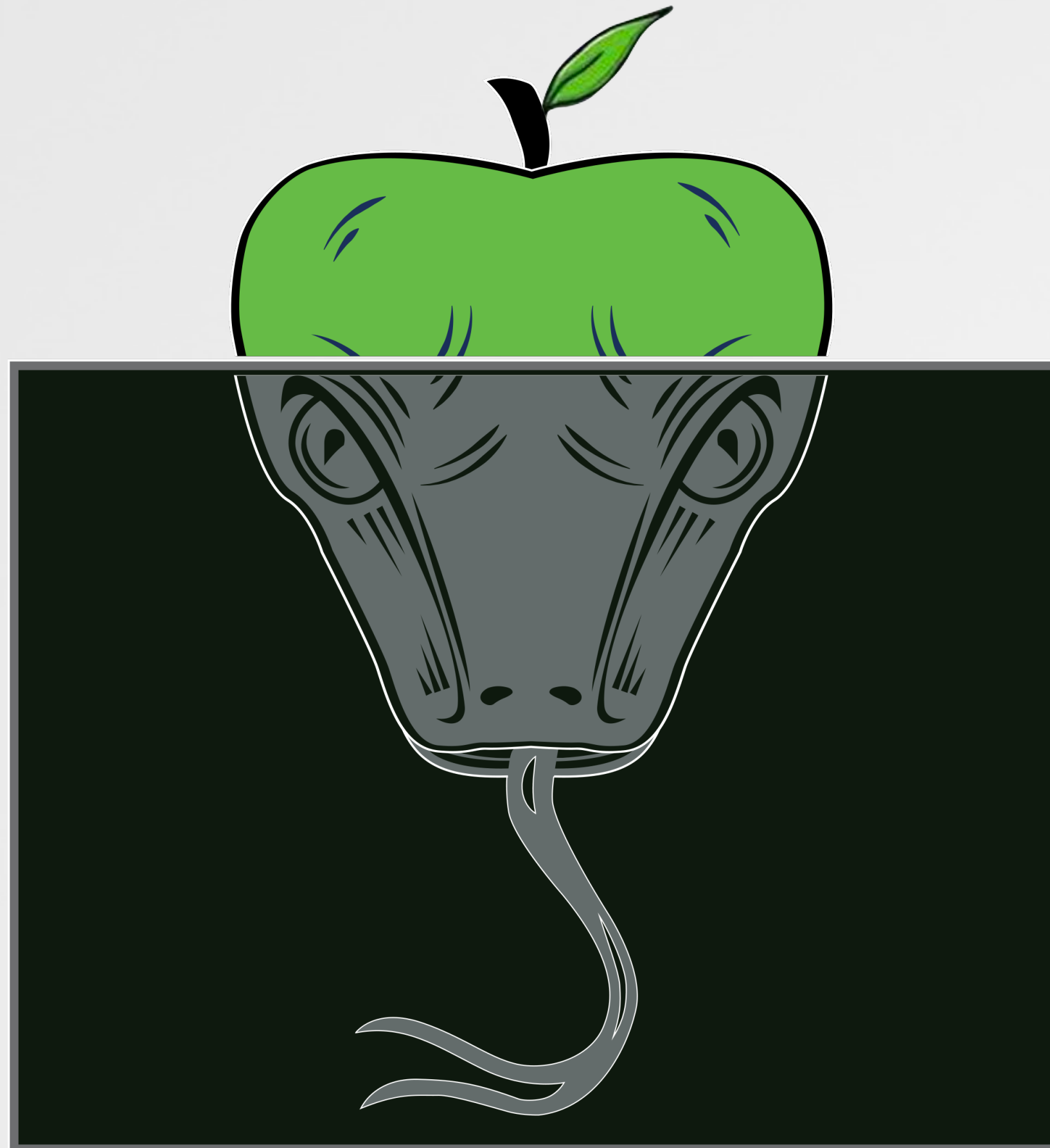
Submissions ⓘ		
Date	Name	Source
2020-12-27 23:37:51	GoSearch22	63b1639b - api

via API
(via KnockKnock)

Autostart Locations ⓘ In-the-wild end-user machine registry keys and autostart locations where this file has been seen.

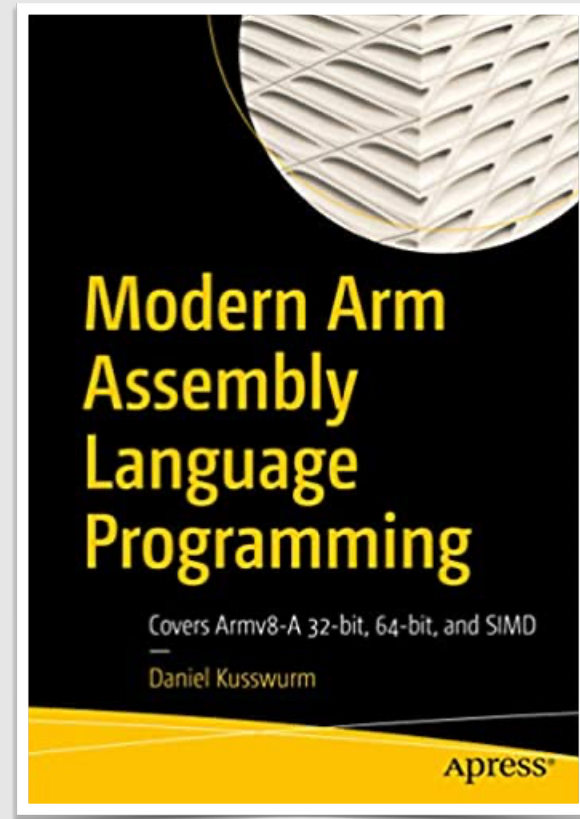
LaunchItems
↳ GoSearch22

Understanding arm64



A BRIEF INTRODUCTION TO ARM64

first, some most excellent resources



"Modern Arm Assembly Language Programming" (Daniel Kusswurm)

free, online

"arm64 Assembly Crash Course"

github.com/Siguza/ios-resources/blob/master/bits/arm64.md



"How to Read ARM64 Assembly Language"

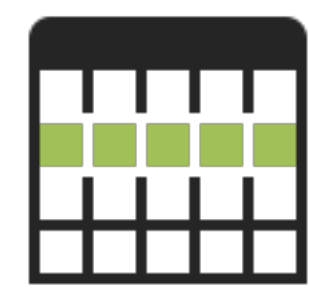
wolchok.org/posts/how-to-read-arm64-assembly-language/

"Introduction To Arm Assembly Basics"

azeria-labs.com/writing-arm-assembly-part-1/

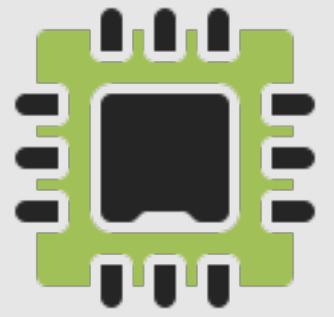
REGISTERS and their uses

(somewhat) synonymous to variables in your fav. programming language

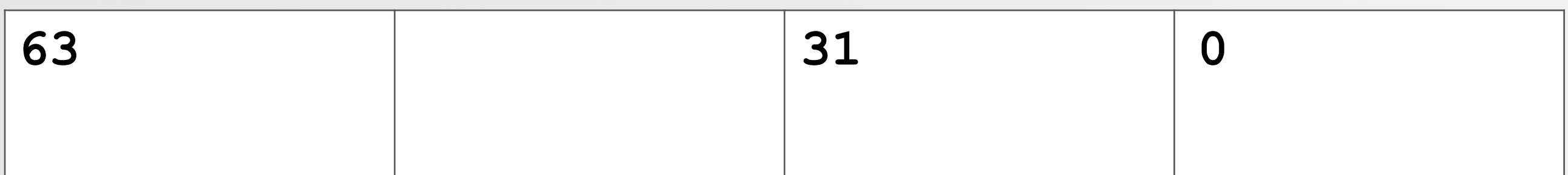


Registers: temporary storage "slots" on the CPU that can be referenced by name.

arm64



31 64-bit registers: x0 - x30
w* (e.g. w0)



PSTATE
(processor state)



Condition flags

- N: negative
- Z: zero
- C: carry
- V: overflow

- sp: stack pointer
- pc: program counter
- xzr: virtual register, value: 0

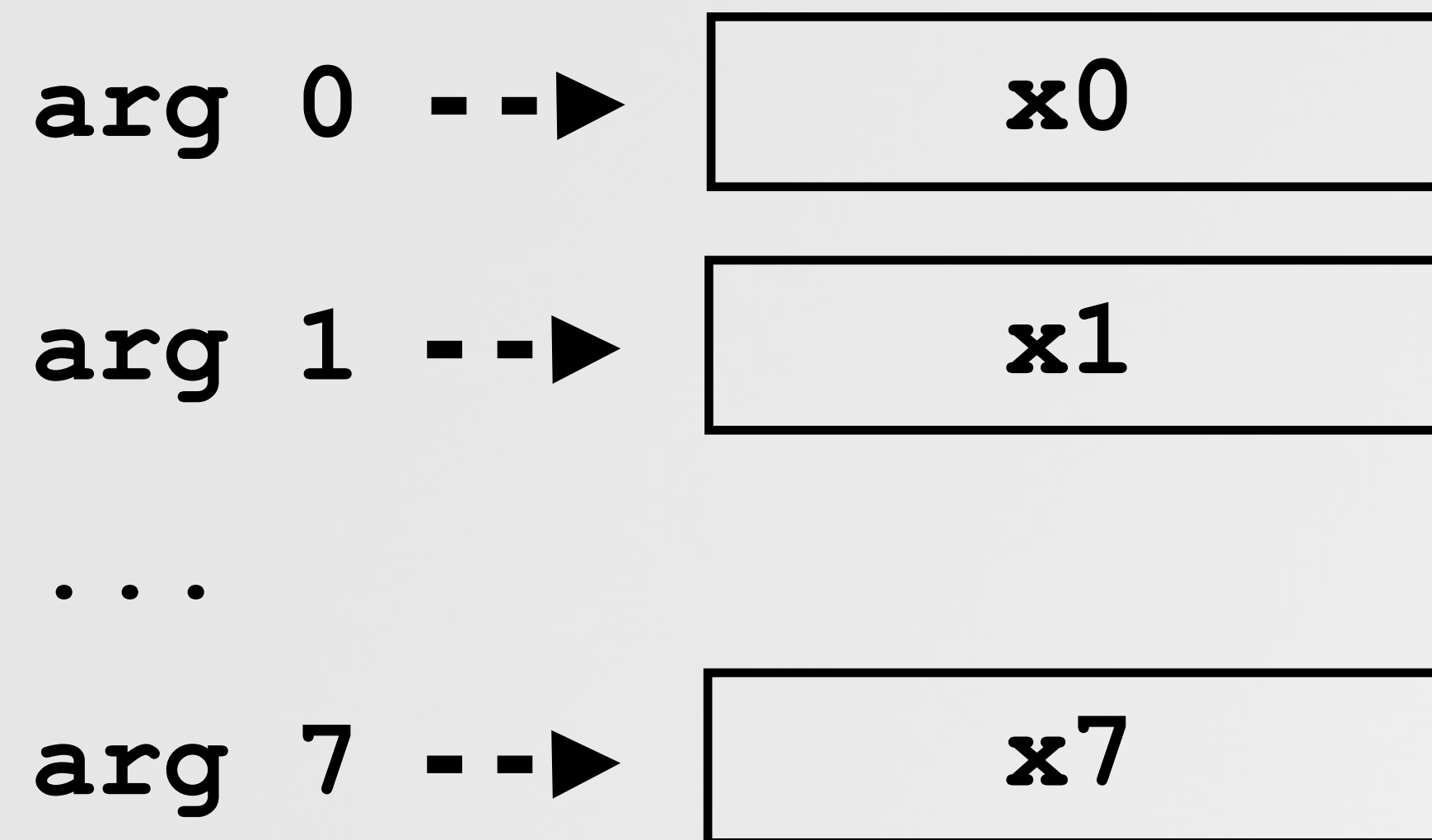
REGISTERS

usage, during a function call

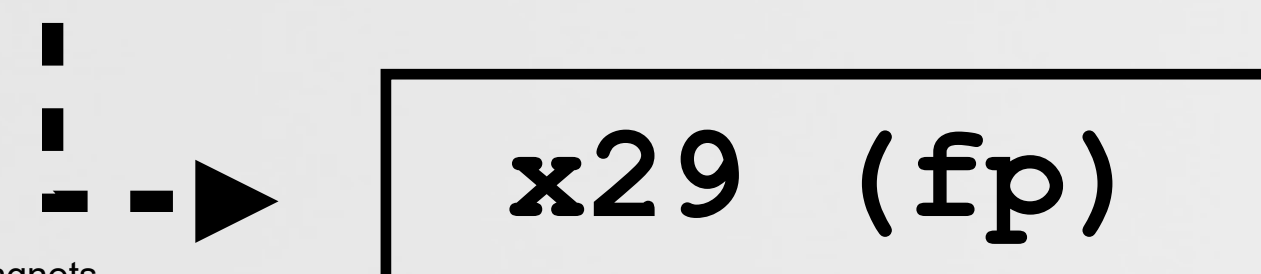


During analysis, we largely focus on api calls and their arguments.

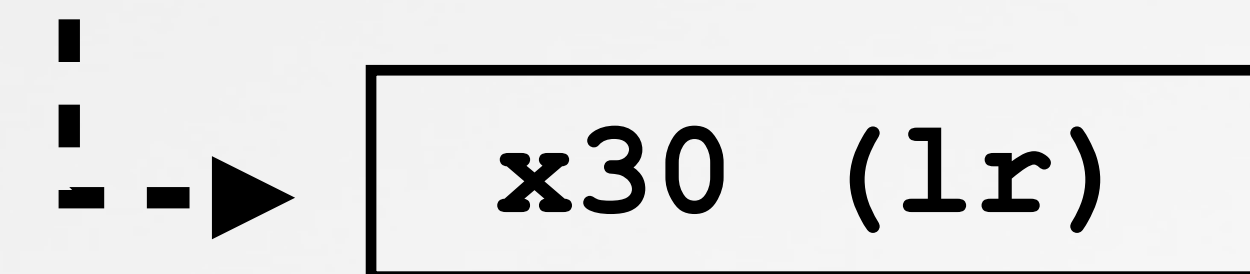
Arguments:



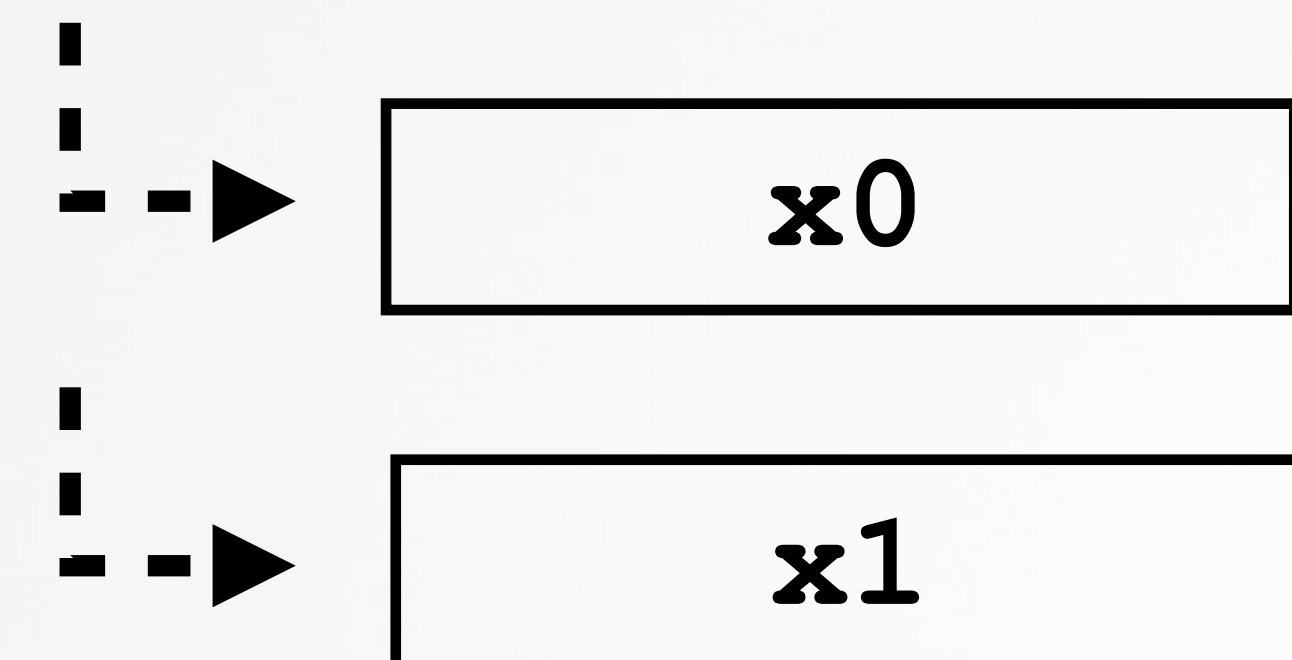
Frame pointer



Return address:



Return value:
(64/128 bits)



INSTRUCTIONS

instruct the cpu what to do



Instructions: map to a specific sequence of bytes that instructs the CPU to perform an operation.



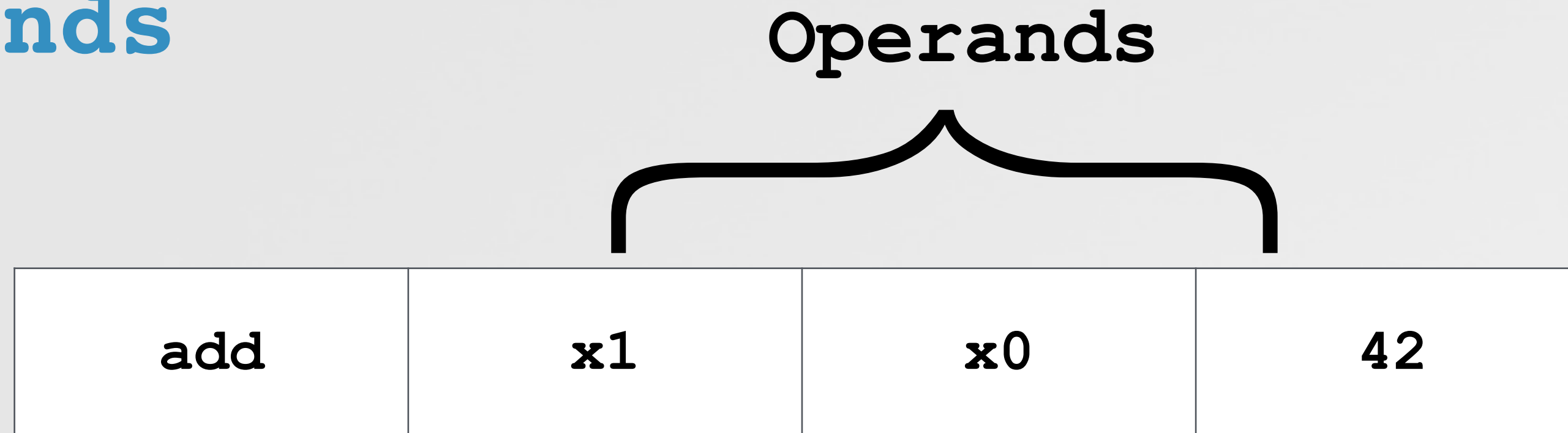
! - - ▶

Mnemonic:

a (human-readable) abbreviation of the operation that the instructions perform.

INSTRUCTIONS

the operands

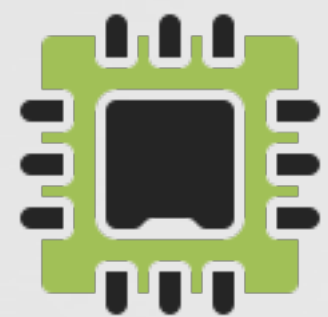


---► 1st register
(usually) destination

Operand types:



Immediate:
a constant value (e.g. 42)



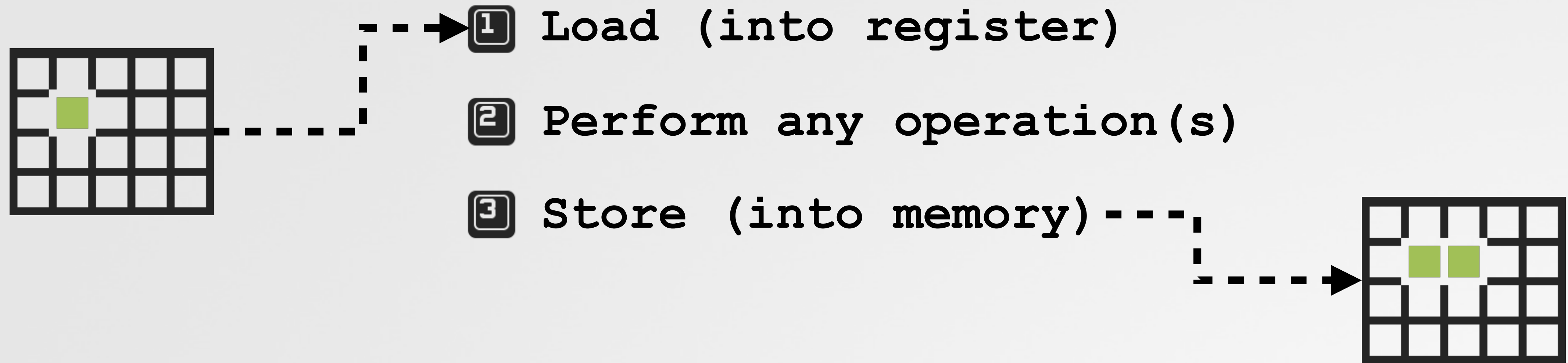
Register:
a cpu register (e.g. x0, x1)



Memory:
a cpu register, that points to a value in memory

MEMORY ACCESS MODEL

arm's model is a "load & store"



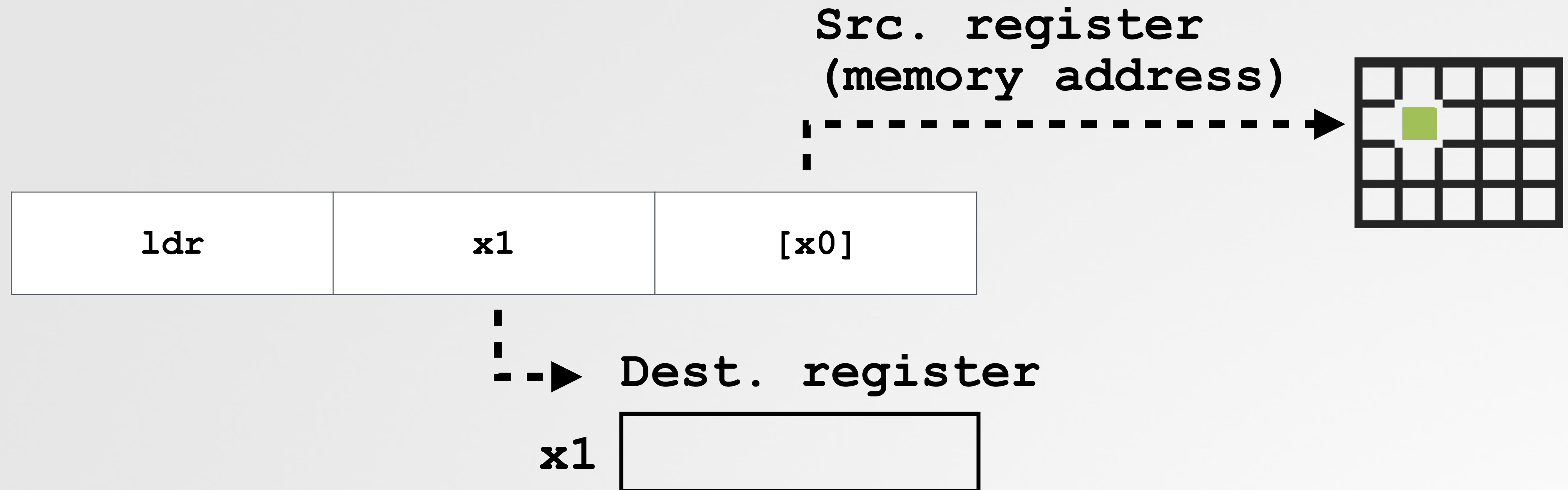
"ARM uses a load-store model for memory access which means that only load/store (LDR and STR) instructions can access memory."

...on ARM data must be moved from memory into registers before being operated on" -Maria Markstedter (Azeria Labs)

...a few other variants, ldp/stp

MEMORY ACCESS MODEL

load via the ldr instruction (+ variants)

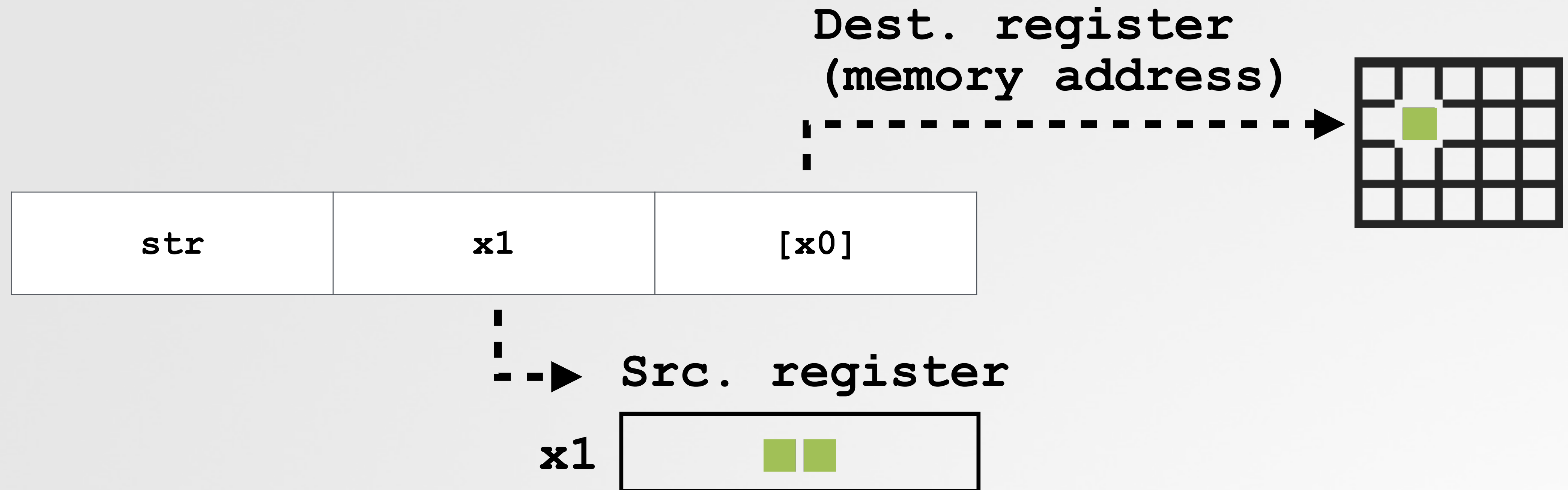


Analogous statement (in C):

```
x1 = *x0;
```

MEMORY ACCESS MODEL

store via the str instruction (+ variants)



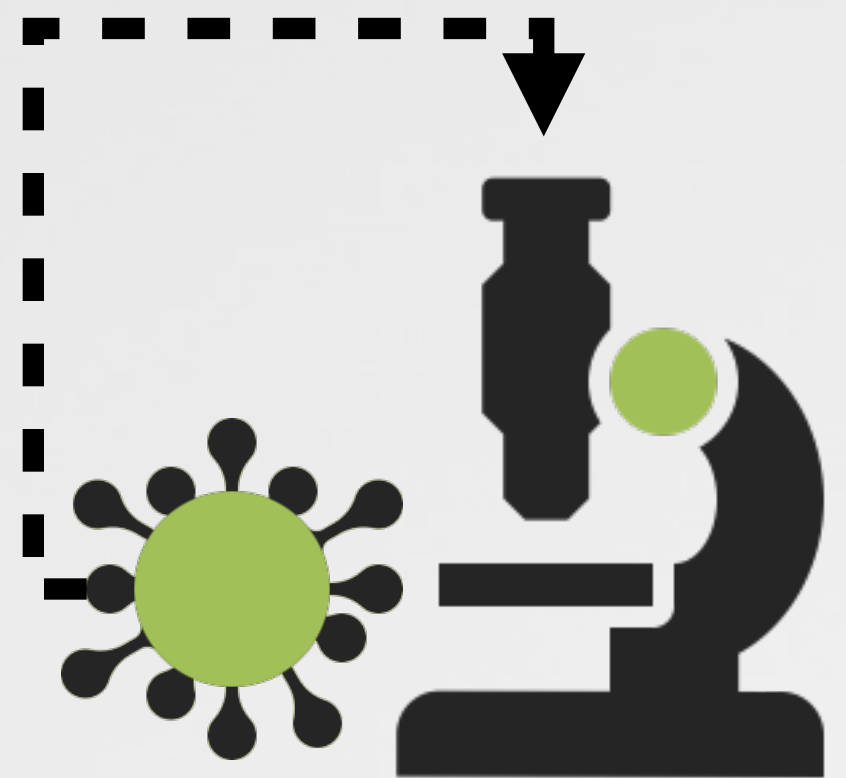
Analogous statement (in C):

```
*x0 = x1;
```

CONDITIONS

set via `cmp`, etc...

`if(isDebugged)`
`exit`



<code>cmp</code>	<code>x0</code>	<code>42</code>
------------------	-----------------	-----------------

`--▶ (discarded) subtract
updates PSTATE flags`

e.g. `x0` is `42`?
`Z` flag is set

PSTATE
(processor state)

<code>N</code>	<code>Z</code>	<code>C</code>	<code>V</code>	...
----------------	----------------	----------------	----------------	-----

`--▶ Condition flags`

CONDITIONS

condition codes



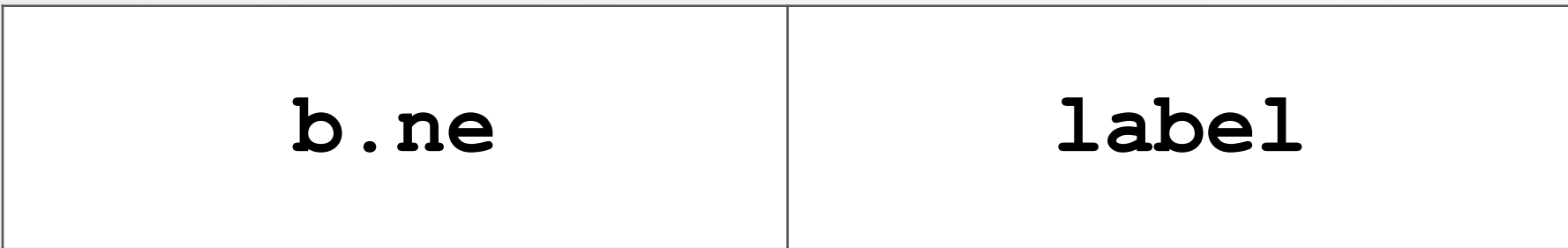
Once (condition) flags have been set, subsequent instructions can act upon them using condition codes

Name	Meaning
EQ	equal
NE	not equal
GE	greater or equal
GT	greater than
LE	lesser or equal
LT	less than
...	

Condition codes

```
01  bl    amBeingDebugged
02  cmp  w0, #1
03  b.ne continue
04  movn w0, #0x0
05  bl    exit
06
07  continue:
08  ...
```

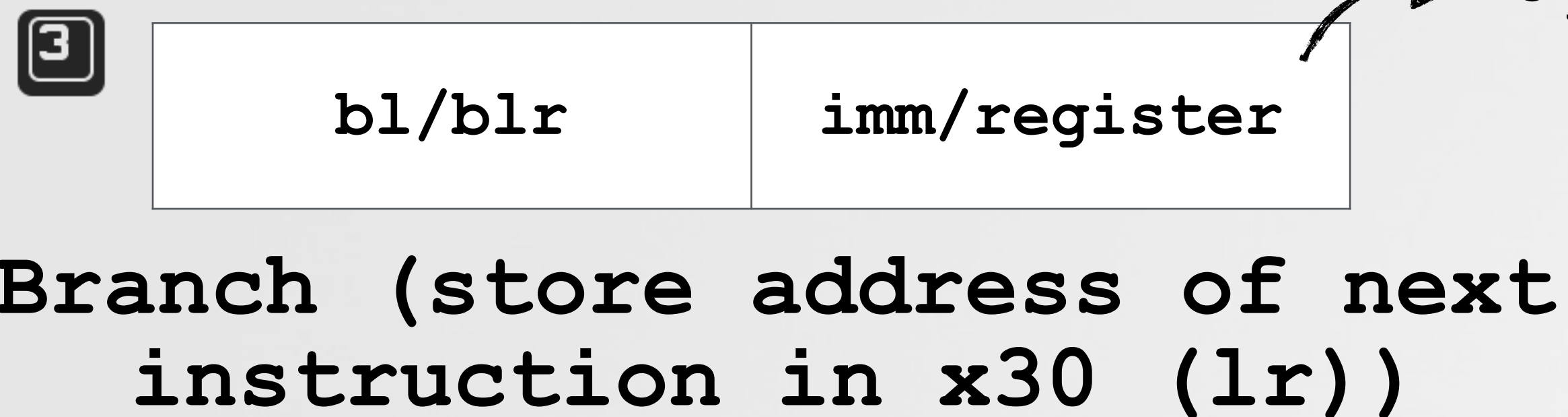
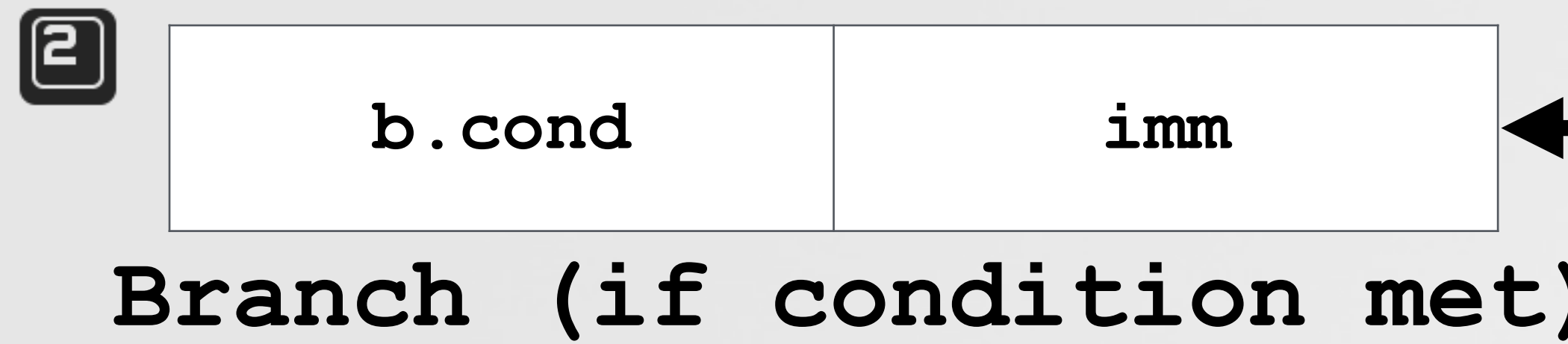
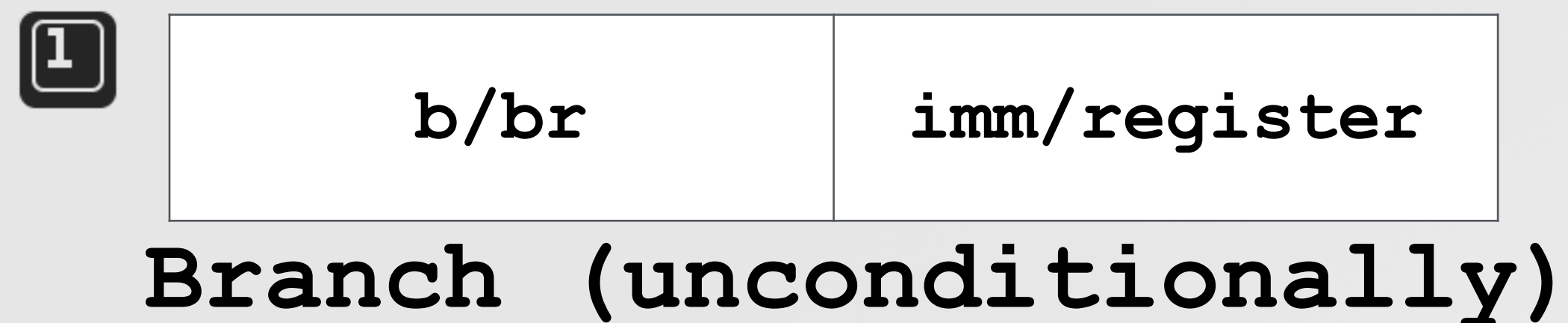
exit if debugged



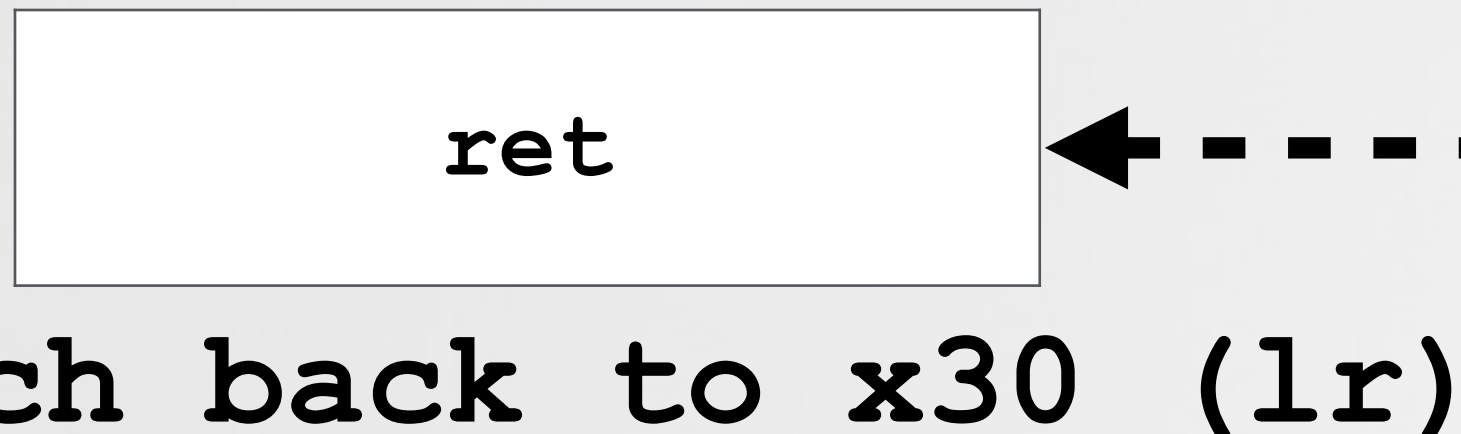
Branch (jump), if Z not set

BRANCHES

alter control flow of a program



e.g. function call



```
01 bl    amBeingDebugged
02 cmp  w0, #1
03 b.ne continue
04 movn w0, #0x0
05 bl   exit
06
07 continue:
08 ...
```

```
01 bl    amBeingDebugged
02 cmp  w0, #1
03 ...
```

&next instruction
(cmp w0, #1)

x30 (lr)

REVERSING "HELLO, WORLD!"

macOS arm64 version

```
01 int main(int argc, char * argv[]) {
02     @autoreleasepool {
03         NSLog(@"Hello, World!");
04     }
05     return 0;
06 }
```

(Apple's) "Hello, World!"

Note, @autoreleasepool:



1 objc_autoreleasePoolPush
+
2 objc_autoreleasePoolPop

```
01 main:
02     sub     sp, sp, #0x30
03     stp    x29, x30, [sp, #0x20]
04     add    x29, sp, #0x20
05     movz   w8, #0x0
06     stur   wzr, [x29, #-0x4]
07     stur   w0, [x29, #-0x8]
08     str    x1, [sp, #0x10]
09     str    w8, [sp, #0xc]
10     bl    objc_autoreleasePoolPush
11     adrp   x9, #0x0000000100004000
12     add    x9, x9, #0x8 ; @"Hello, World!"
13     str    x0, [sp]
14     mov    x0, x9
15     bl    NSLog
16     ldr    x0, [sp]
17     bl    objc_autoreleasePoolPop
18     ldr    w0, [sp, #0xc]
19     ldp    x29, x30, [sp, #0x20]
20     add    sp, sp, #0x30
21     ret
```

"Hello, World!"
disassembled

REVERSING "HELLO, WORLD!"

function prologue

```

01  main:
02  sub   sp, sp, #0x30
03  stp   x29, x30, [sp, #0x20]
04  add   x29, sp, #0x20
05  movz  w8, #0x0
06  stur  wzr, [x29, #-0x4]
07  stur  w0, [x29, #-0x8]
08  str   x1, [sp, #0x10]
09  str   w8, [sp, #0xc]
10  bl    objc_autoreleasePoolPush
11  adrp  x9, #0x0000000100004000
12  add   x9, x9, #0x8 ; @"Hello, World!"
13  str   x0, [sp]
14  mov   x0, x9
15  bl    NSLog
16  ldr   x0, [sp]
17  bl    objc_autoreleasePoolPop
18  ldr   w0, [sp, #0xc]
19  ldp   x29, x30, [sp, #0x20]
20  add   sp, sp, #0x30
21  ret

```

- - - - -> Subtract 0x30 from stack pointer
- - - - -> Store x29 & x30 at SP + 0x20
- - - - -> Set frame pointer (x29) to sp + 0x20
- - - - -> Save registers/init local variables

	Offset	Value
sp	0x30	
	0x28	x30
x29	0x20	x29
	...	
sp	0x00	



Function prologue makes space on the stack for saving registers, local variables, and init's frame pointer

REVERSING "HELLO, WORLD!"

invoking objc_autoreleasePoolPush

```
01  main:
02  sub    sp, sp, #0x30
03  stp    x29, x30, [sp, #0x20]
04  add    x29, sp, #0x20
05  movz   w8, #0x0
06  stur   wzr, [x29, #-0x4]
07  stur   w0, [x29, #-0x8]
08  str    x1, [sp, #0x10]
09  str    w8, [sp, #0xc]
10  bl     objc_autoreleasePoolPush
11  adrp   x9, #0x0000000100004000
12  add    x9, x9, #0x8 ; @"Hello, World!"
13  str    x0, [sp]
14  mov    x0, x9
15  bl     NSLog
16  ldr    x0, [sp]
17  bl     objc_autoreleasePoolPop
18  ldr    w0, [sp, #0xc]
19  ldp    x29, x30, [sp, #0x20]
20  add    sp, sp, #0x30
21  ret
```

Branch to (call)
objc_autoreleasePoolPush

address of next instruction,
stored in link register (x30)

Return value (x0: pool object)
saved to local variable



objc_autoreleasePoolPush takes no arguments
...returns a pointer to a pool object (in x0).

REVERSING "HELLO, WORLD!"

invoking NSLog with the "Hello, World!" string

```
01 main:
02 sub    sp, sp, #0x30
03 stp    x29, x30, [sp, #0x20]
04 add    x29, sp, #0x20
05 movz   w8, #0x0
06 stur   wzr, [x29, #-0x4]
07 stur   w0, [x29, #-0x8]
08 str    x1, [sp, #0x10]
09 str    w8, [sp, #0xc]
10 bl     objc_autoreleasePoolPush
11 adrp   x9, #0x0000000100004000
12 add    x9, x9, #0x8 ; @"Hello, World!"
13 str    x0, [sp]
14 mov    x0, x9
15 bl     NSLog
16 ldr    x0, [sp]
17 bl     objc_autoreleasePoolPop
18 ldr    w0, [sp, #0xc]
19 ldp    x29, x30, [sp, #0x20]
20 add    sp, sp, #0x30
21 ret
```

Initialize address to
"Hello World!" (string) object

Initialize 1st argument with
address of string object

Branch to (call) NSLog function



Here, NSLog is invoked with a single argument, the address of the string object to print (passed in x0).

REVERSING "HELLO, WORLD!"

invoking objc_autoreleasePoolPop with pool object

```
01 main:
02 sub    sp, sp, #0x30
03 stp    x29, x30, [sp, #0x20]
04 add    x29, sp, #0x20
05 movz   w8, #0x0
06 stur   wzr, [x29, #-0x4]
07 stur   w0, [x29, #-0x8]
08 str    x1, [sp, #0x10]
09 str    w8, [sp, #0xc]
10 bl     objc_autoreleasePoolPush
11 adrp   x9, #0x0000000100004000
12 add    x9, x9, #0x8 ; @"Hello, World!"
13 str    x0, [sp]
14 mov    x0, x9
15 bl     NSLog
16 ldr    x0, [sp]
17 bl     objc_autoreleasePoolPop
18 ldr    w0, [sp, #0xc]
19 ldp    x29, x30, [sp, #0x20]
20 add    sp, sp, #0x30
21 ret
```

Initialize 1st argument with
address pool object (previous
stored on the stack)

Branch to (call)
objc_autoreleasePoolPop function

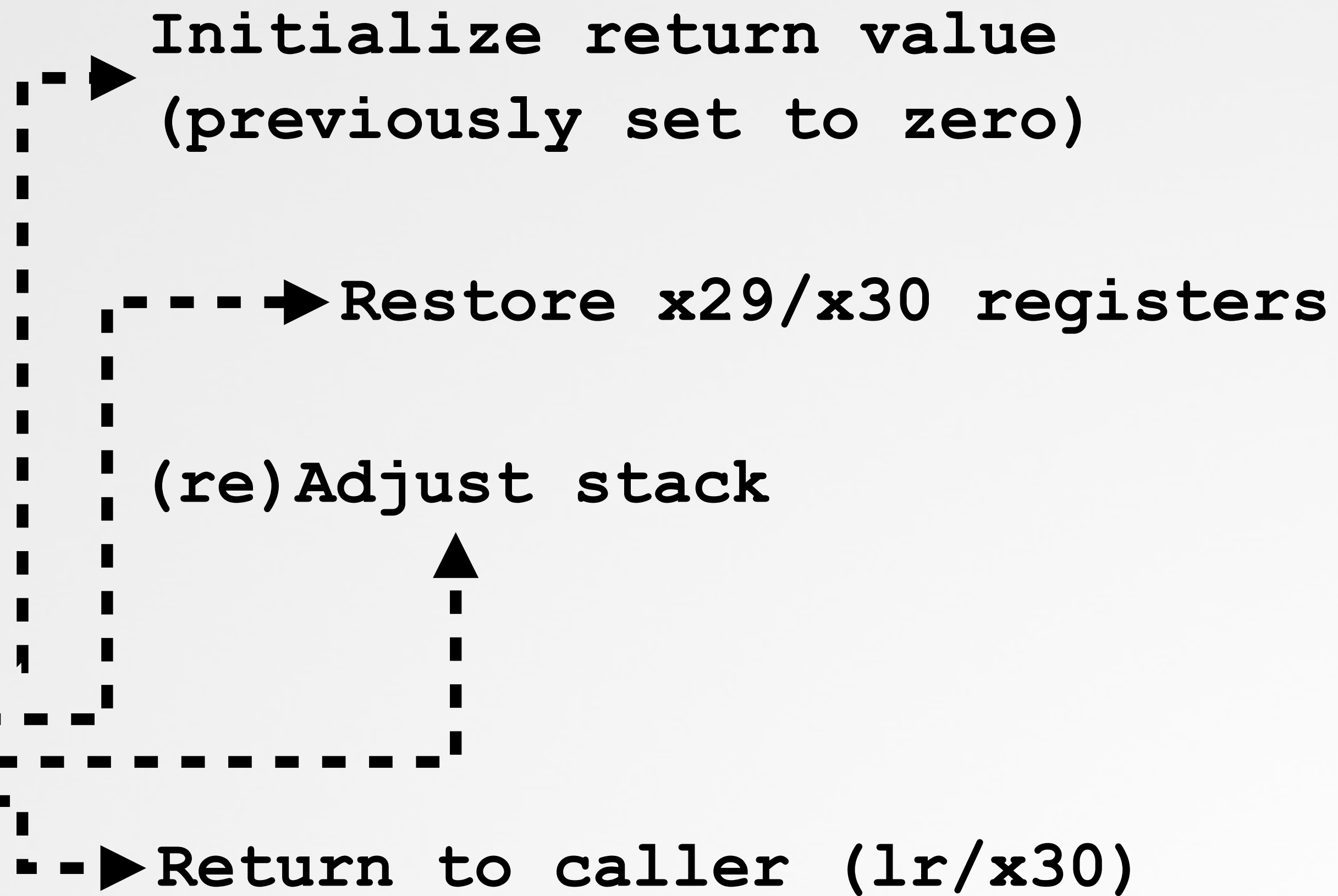


objc_autoreleasePoolPop takes a single argument, the
address of the pool object to release (passed in x0).

REVERSING "HELLO, WORLD!"

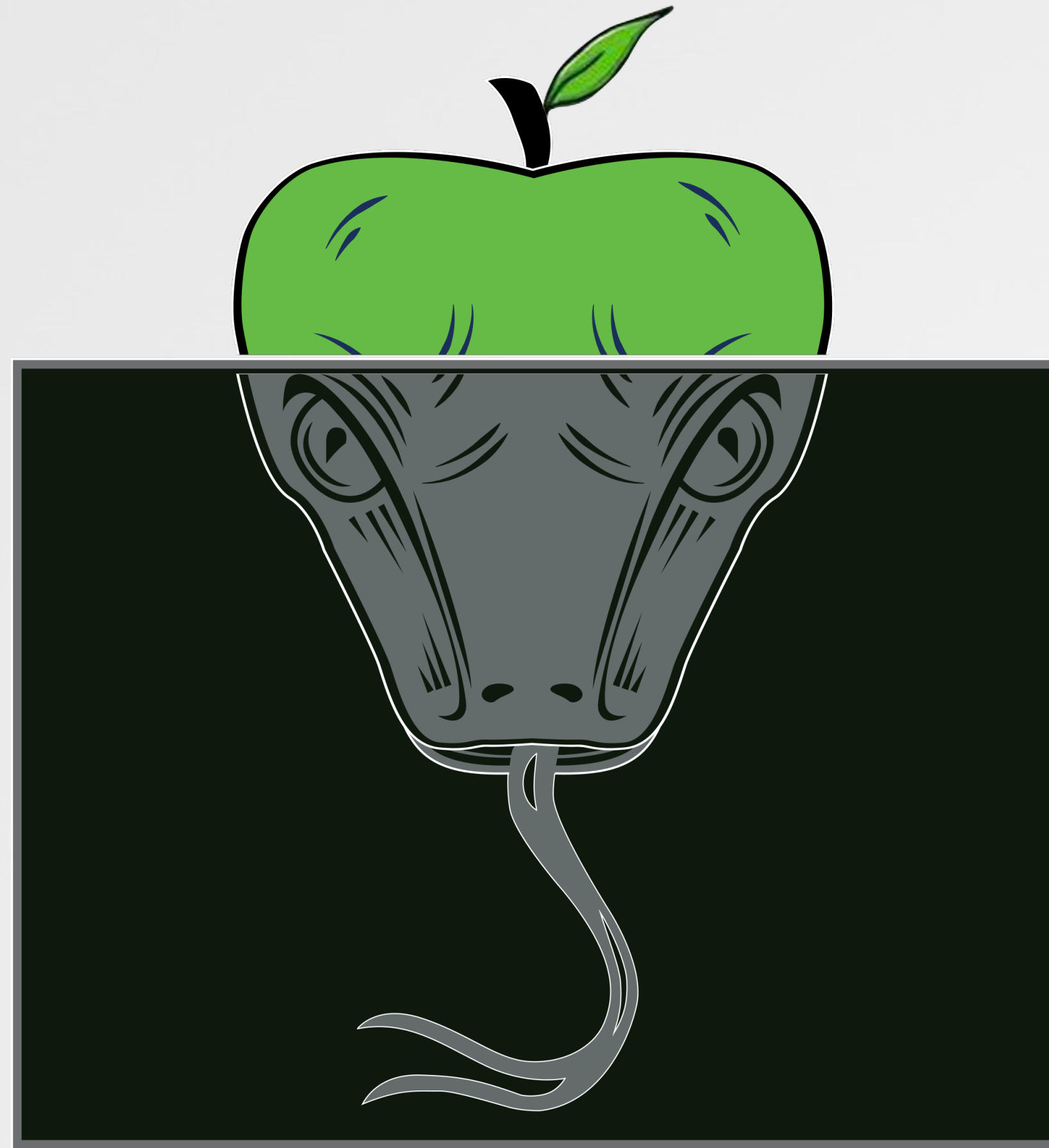
function epilogue

```
01 main:
02 sub sp, sp, #0x30
03 stp x29, x30, [sp, #0x20]
04 add x29, sp, #0x20
05 movz w8, #0x0
06 stur wzr, [x29, #-0x4]
07 stur w0, [x29, #-0x8]
08 str x1, [sp, #0x10]
09 str w8, [sp, #0xc]
10 bl objc_autoreleasePoolPush
11 adrp x9, #0x0000000100004000
12 add x9, x9, #0x8 ; @"Hello, World!"
13 str x0, [sp]
14 mov x0, x9
15 bl NSLog
16 ldr x0, [sp]
17 bl objc_autoreleasePoolPop
18 ldr w0, [sp, #0xc]
19 ldp x29, x30, [sp, #0x20]
20 add sp, sp, #0x30
21 ret
```



A function epilogue restores saved registers, and (re)adjusts the stack. Return, branches to lr (x30).

Practical M1 Malware Analysis



A FUNDAMENTAL UNDERSTANDING OF ARM64 SUFFICE?

...often, yes!

```
01 main:
02 sub    sp, sp, #0x30
03 stp   x29, x30, [sp, #0x20]
04 add   x29, sp, #0x20
05 movz  w8, #0x0
06 stur  wzr, [x29, #-0x4]
07 stur  w0, [x29, #-0x8]
08 str   x1, [sp, #0x10]
09 str   w8, [sp, #0xc]
10 bl    objc_autoreleasePoolPush
11 adrp  x9, #0x0000000100004000
12 add   x9, x9, #0x8 ; @"Hello, World!"
13 str   x0, [sp]
14 mov   x0, x9
15 bl    NSLog
16 ldr   x0, [sp]
17 bl    objc_autoreleasePoolPop
18 ldr   w0, [sp, #0xc]
19 ldp   x29, x30, [sp, #0x20]
20 add   sp, sp, #0x30
21 ret
```

```
01 int main(int arg0, int arg1) {
02     var_20 = objc_autoreleasePoolPush();
03     NSLog(@"Hello, World!");
04     objc_autoreleasePoolPop(var_20);
05     return 0x0;
06 }
```

"Hello, World!" decompiled

Dynamic Analysis Tools:



File monitor



Process monitor



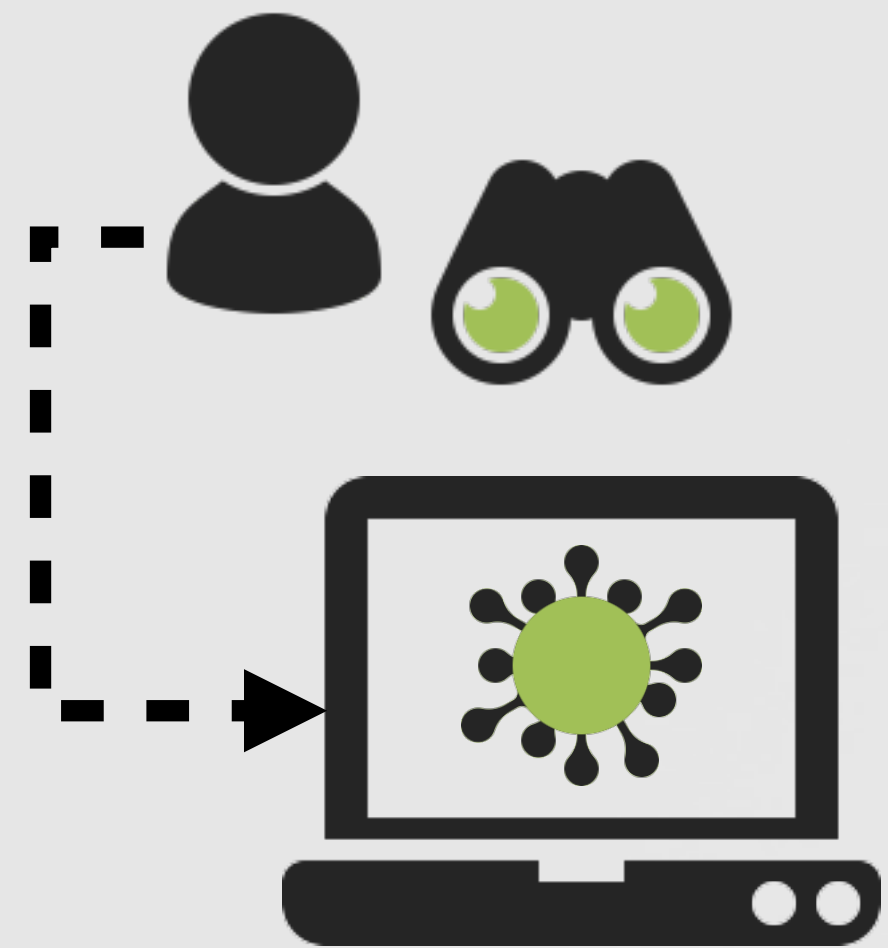
Network monitor



By leveraging a decompiler and dynamic analysis tools, often a fundamental understanding of arm64 will suffice!

DYNAMIC ANALYSIS TOOLS

may (trivially) reveal malware's capabilities



Analysis
machine

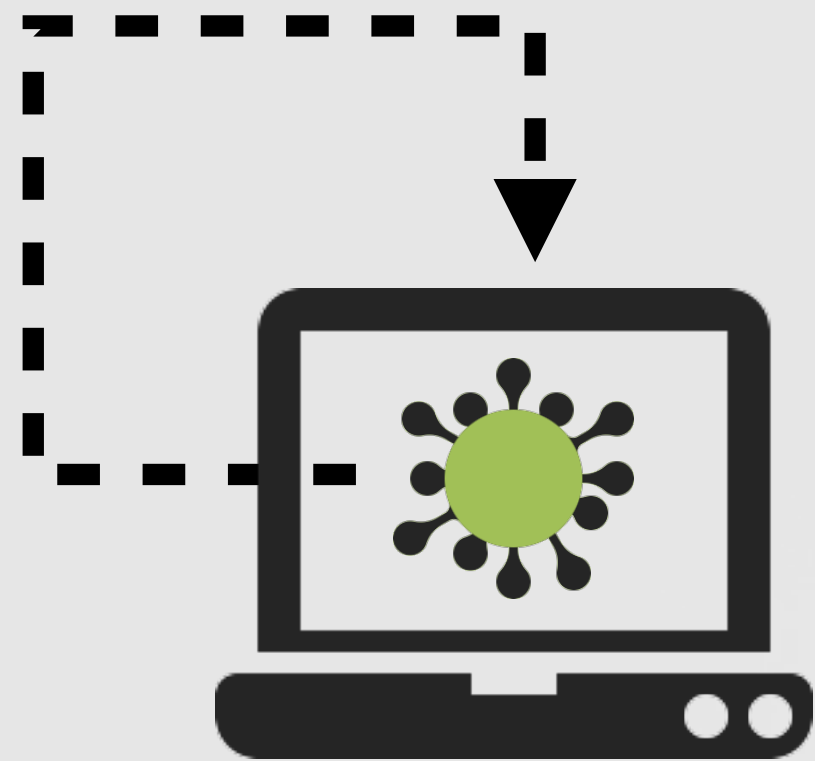
```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/Library/LaunchAgents/mdworker.plist",
    "process" : {
      "uid" : 501,
      "arguments" : [
        "/bin/sh",
        "-c",
        "/Users/user/Desktop/eTrader.app/Contents/Utils/mdworker"
      ],
      "path" : "/Users/user/Desktop/eTrader.app/Contents/Utils/mdworker",
      "name" : "mdworker"
    }
  }
}
```

launch agent persistence

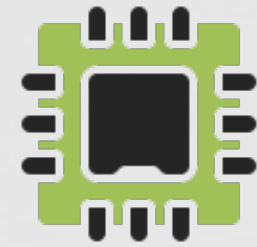
Uncovering persistence
(via a file monitor)

ANTI-ANALYSIS LOGIC

aim to thwart (dynamic) analysis environments/tools



Introspection



Am I being debugged?



Am I in a virtual machine?

```
% llldb GoSearch22.app
(llldb) target create "GoSearch22.app"

(llldb) c
Process 654 resuming
Process 654 exited with status = 45 (0x0000002d)
```

simply terminates :l

GoSearch22 vs. debugger



One must identify and bypass anti-analysis mechanisms before comprehensive analysis of a malicious sample can commence!

GOSEARCH22

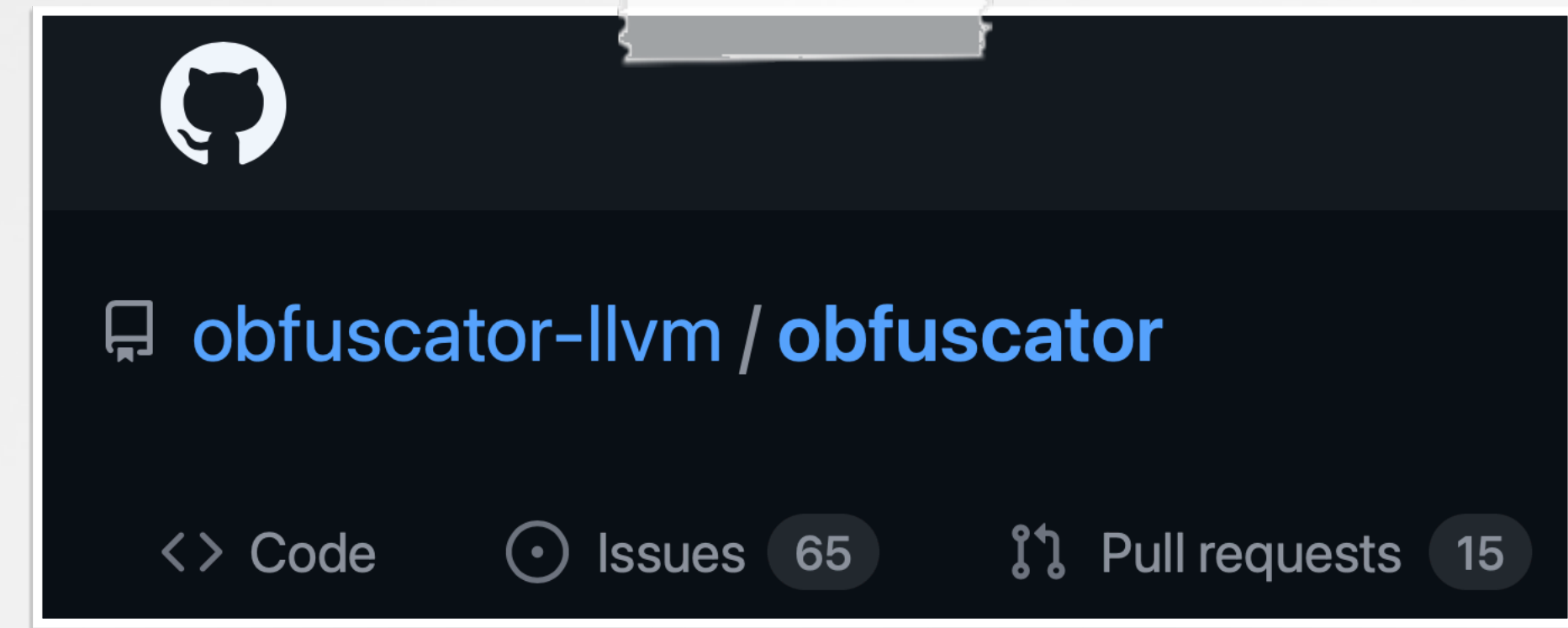
...also contains static analysis obfuscations

```
*(int128_t *)0x100071a30 = *(int128_t *)0x100071a30;
*(int8_t *)0x100071a40 = *(int8_t *)0x100071a40 ^ 0xcd;
*(int128_t *)0x100071a50 = *(int128_t *)0x100071a50;
*(int8_t *)0x100071a60 = *(int8_t *)0x100071a60 ^ 0xa;
*(int8_t *)0x100071a61 = *(int8_t *)0x100071a61 ^ 0x3a;
*(int8_t *)0x100071a62 = *(int8_t *)0x100071a62 ^ 0xaf;
*(int8_t *)0x100071a63 = *(int8_t *)0x100071a63 ^ 0x48;
*(int128_t *)0x100071a70 = *(int128_t *)0x100071a70;
*(int8_t *)0x100071a80 = *(int8_t *)0x100071a80 ^ 0xc9;
*(int128_t *)0x100071a90 = *(int128_t *)0x100071a90;
*(int8_t *)0x100071aa0 = *(int8_t *)0x100071aa0 ^ 0xfffffffffffffffb;
*(int8_t *)0x100071aa1 = *(int8_t *)0x100071aa1 ^ 0xea;
*(int8_t *)0x100071aa2 = *(int8_t *)0x100071aa2 ^ 0xd5;
*(int8_t *)0x100071aa3 = *(int8_t *)0x100071aa3 ^ 0x51;
*(int128_t *)0x100071ab0 = *(int128_t *)0x100071ab0;
*(int8_t *)0x100071ac0 = *(int8_t *)0x100071ac0 ^ 0xd4;
*(int128_t *)0x100071ad0 = *(int128_t *)0x100071ad0;
*(int8_t *)0x100071ae0 = *(int8_t *)0x100071ae0 ^ 0xed;
*(int8_t *)0x100071ae1 = *(int8_t *)0x100071ae1 ^ 0xf2;
*(int8_t *)0x100071ae2 = *(int8_t *)0x100071ae2 ^ 0xa;
*(int8_t *)0x100071ae3 = *(int8_t *)0x100071ae3 ^ 0xffffffffffffff8f;
v0 = v0 ^ v1 ^ v1 ^ v1 ^ v1 ^ v1 ^ v1 ^ v1;
```

```
var_70 = var_78;
var_98 = dlsym(dlopen(0x0, 0xa), 0x100071b40);
dlsym(dlopen(0x0, 0xa), 0x100071b28);
dlsym(dlopen(0x0, 0xa), 0x100071b10);
dlsym(dlopen(0x0, 0xa), 0x100071af0);
dlsym(dlopen(0x0, 0xa), 0x100071ad0);
dlsym(dlopen(0x0, 0xa), 0x100071ab0);
dlsym(dlopen(0x0, 0xa), 0x100071a90);
dlsym(dlopen(0x0, 0xa), 0x100071a70);
```

Spurious function calls

Garbage instructions?



Popular obfuscator



See, "Using LLVM to Obfuscate Your Code During Compilation" (www.apriorit.com)

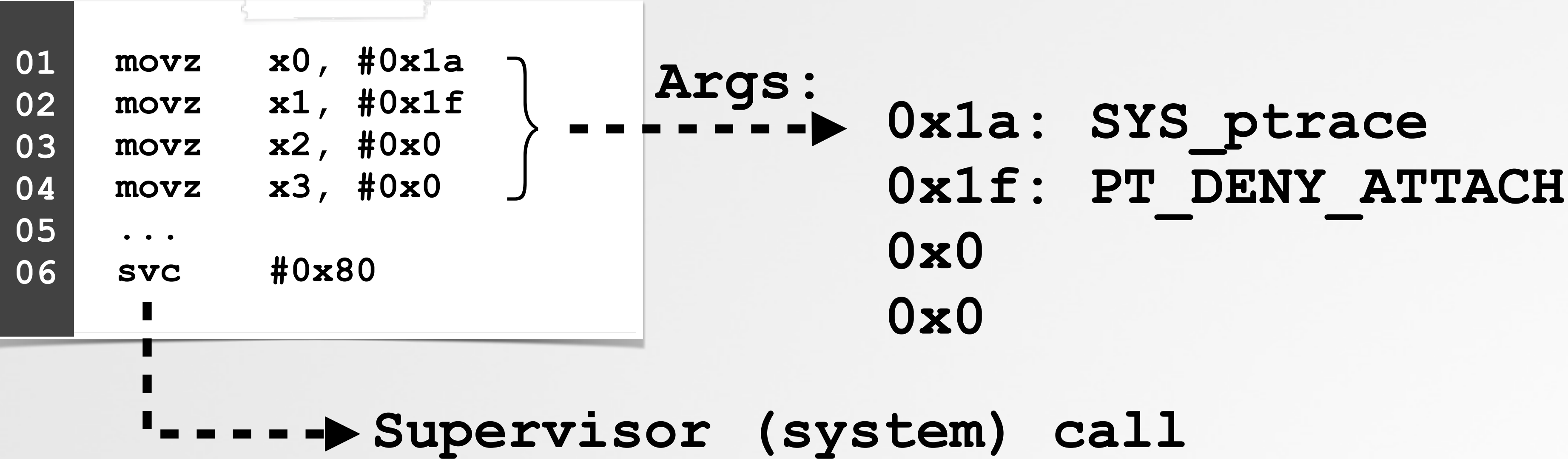
GoSEARCH22 's ANTI-ANALYSIS LOGIC


debugger detection via ptrace/PT_DENY_ATTACH

```
% 1ldb GoSearch22.app  
Process 654 exited with status = 45 (0x0000002d)
```

45 (02xd)
ENOTSUP (from PT_DENY_ATTACH)

GoSearch22 vs. debugger



 ptrace() + PT_DENY_ATTACH, prevents future attachments or terminates (with 45) if a debugger is currently attached.

BYPASSING ANTI-ANALYSIS LOGIC

once detected and identified, trivial to bypass

```
01 0x000000001000541f4 movz    x3, #0x0
02 0x000000001000541f8 movz    x16, #0x0
03 0x000000001000541fc svc    #0x80
04
05 0x00000000100054200 movz    w11, #0x6b8f
```

simply skip over ptrace system call :)

```
% lldb GoSearch22.app

(lldb) b 0x000000001000541fc
Breakpoint 1: address = 0x000000001000541fc

(lldb) Process 1486 stopped
* thread #1, queue = 'com.apple.main-thread'
  stop reason = breakpoint 1.1:
-> 0x000000001000541fc svc    #0x80
(lldb) reg write $pc 0x100054200
```

modify PC

Modify pc register

GOSEARCH22'S ANTI-ANALYSIS LOGIC

system integrity protection (sip) status detection

```
01 ldr x8, [sp, #0x190 + var_120]
02 ldr x0, [sp, #0x190 + var_100]
03 ldr x1, [sp, #0x190 + var_F8]
04 blr x8
```

Two arguments
...but what's the branch target?

```
% llldb GoSearch22.app
...
(lldb) x/i $pc
-> 0x1000538dc: 0xd63f0100 blr x8

(lldb) reg read $x8
x8 = 0x000000193a5f160 libobjc.A.dylib`objc_msgSend
```

call to objc_msgSend

Debugger introspection



As we've identified (and thwarted) the malware's anti-debugging logic, we can now fully leverage the debugger!

GOSEARCH22'S ANTI-ANALYSIS LOGIC

system integrity protection (sip) status detection

objc_msgSend

Sends a message with a simple return value to an instance of a class.

Declaration

```
void objc_msgSend(void);
```

Parameters

self

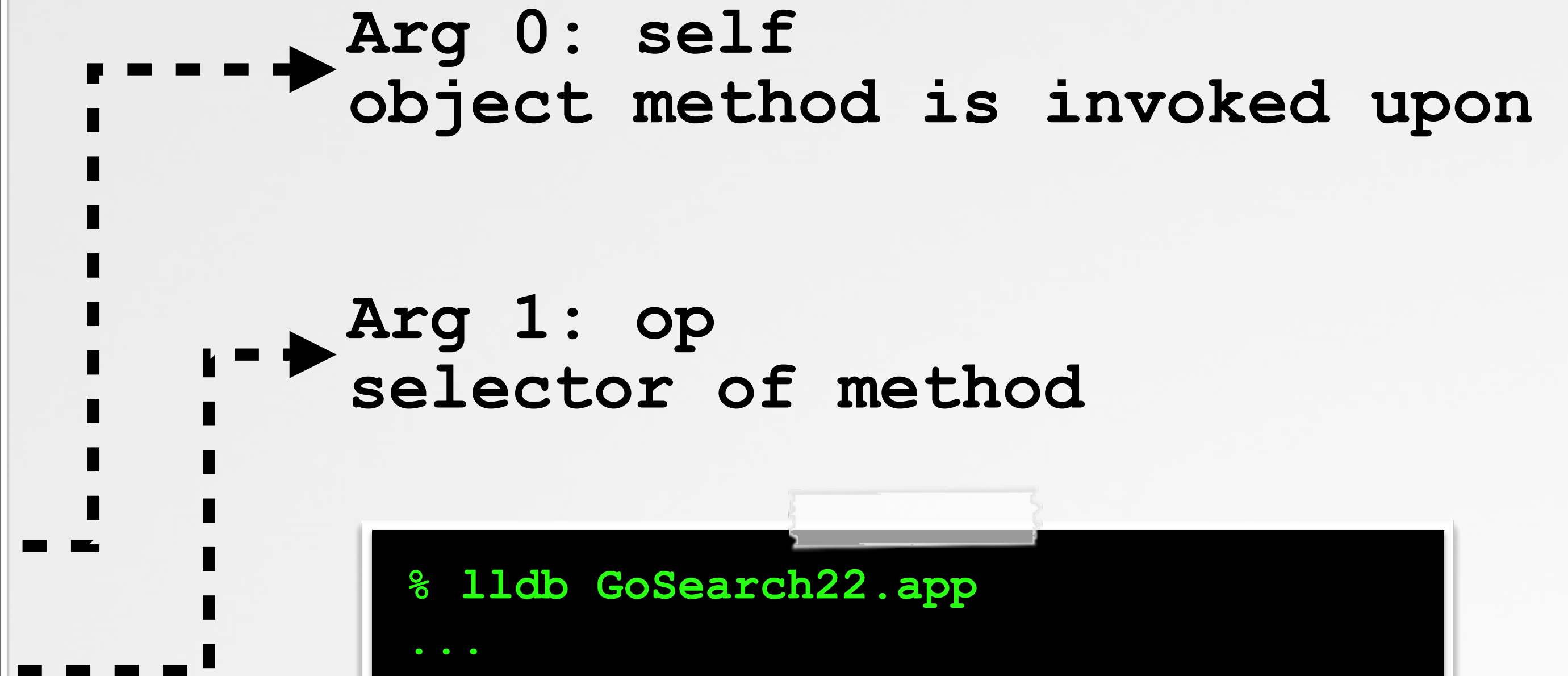
A pointer that points to the instance of the class that is to receive the message.

op

The selector of the method that handles the message.

...

A variable argument list containing the arguments to the method.



```
% llldb GoSearch22.app
...

(lldb) po $x0
<NSConcreteTask: 0x1058306c0>

(lldb) x/s $x1
0x1e9fd4fae: "launch"
```

Debugger introspection
[NSTask launch];

GOSEARCH22 's ANTI-ANALYSIS LOGIC

SIP status detection

Class
NSTask
An object representing a subprocess of the current process.

Instance Property
launchPath
Sets the receiver's executable.

Instance Property
arguments
Sets the command arguments that should be used to launch the executable.

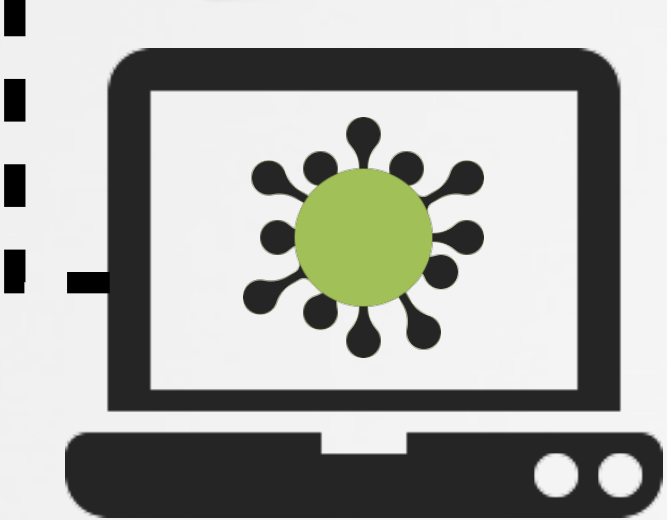
NSTask
+ it's properties

```
(lldb) po [$x0 launchPath]
/bin/sh

(lldb) po [$x0 arguments]
<__NSArrayI 0x10580dfd0>(
  -c,
  command -v csrutil > /dev/null && csrutil status |
  grep -v "enabled" > /dev/null && echo 1 || echo 0
)
```

SIP status detection
(via "csrutil status")

```
% csrutil status
System Integrity Protection status: disabled.
```



SIP: disabled?
(malware **exits!)**

analysis machine

GOSEARCH22'S ANTI-ANALYSIS LOGIC

virtual machine detection

```
01 ldr x8, [sp, #0x190 + var_120]
02 ldr x0, [sp, #0x190 + var_100]
03 ldr x1, [sp, #0x190 + var_F8]
04 blr x8
```

(another) call to obj_msgSend

```
(lldb) po $x0
<NSConcreteTask: 0x1058306c0>
```

```
(lldb) po [$x0 launchPath]
/bin/sh
```

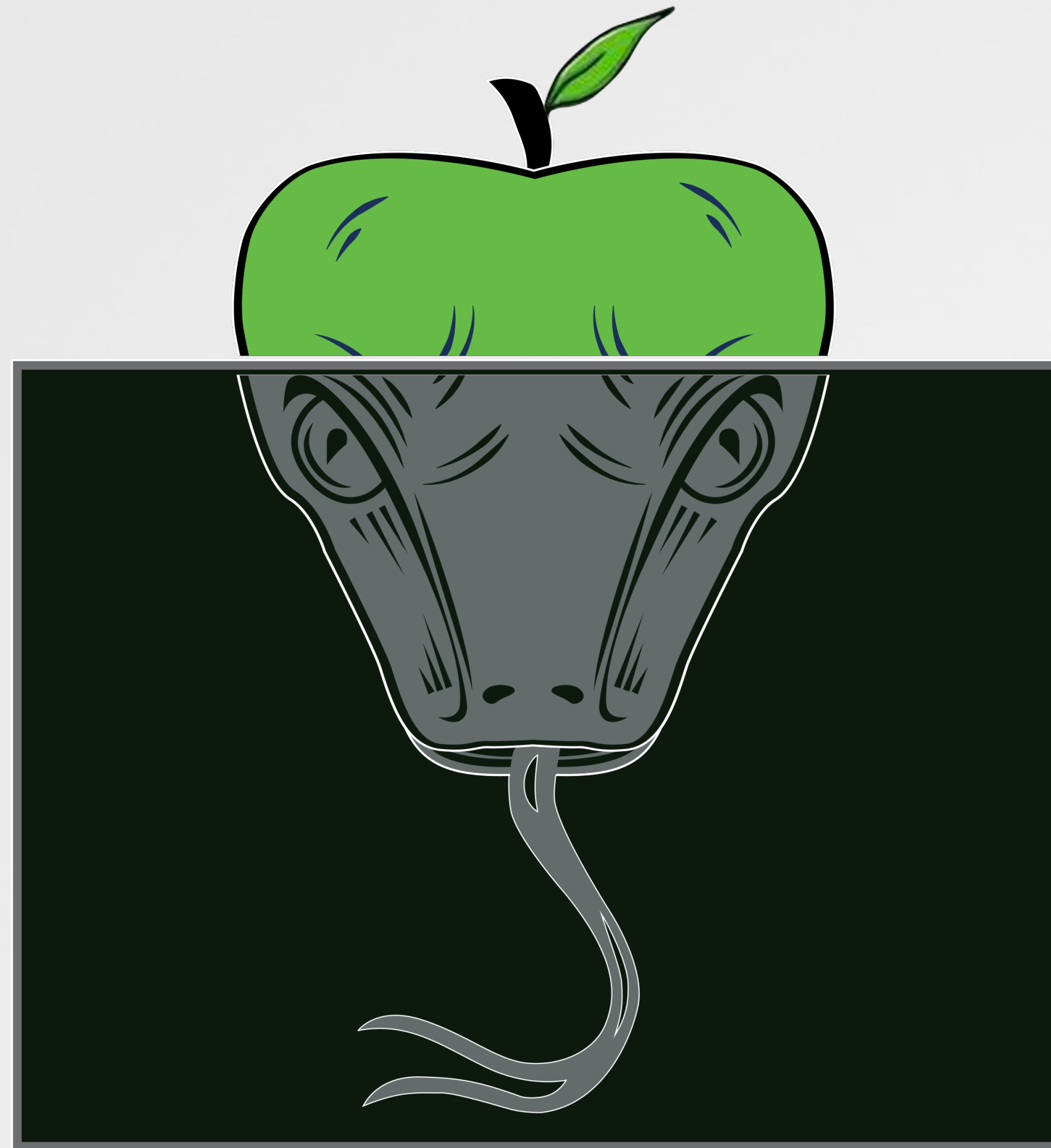
```
(lldb) po [$x0 arguments]
<__NSArrayI 0x10580c1f0> (
```

```
-c,
readonly VM_LIST="VirtualBox\|Oracle\|VMware\|Parallels\|qemu";is_hwmodel_vm(){ ! sysctl -n hw.model|grep
"Mac">/dev/null;};is_ram_vm(){(($(($sysctl -n hw.memsize)/ 1073741824))<4)};is_ped_vm(){ local -r ped=$
(ioreg -rd1 -c IOPlatformExpertDevice);echo "${ped}"|grep -e "board-id" -e "product-name" -e "model"|grep
-qi "${VM_LIST}"||echo "${ped}"|grep "manufacturer"|grep -v "Apple">/dev/null;};is_vendor_name_vm(){ ioreg
-l|grep -e "Manufacturer" -e "Vendor Name"|grep -qi "${VM_LIST}";};is_hw_data_vm(){ system_profiler
SPHardwareDataType 2>&1 /dev/null|grep -e "Model Identifier"|grep -qi "${VM_LIST}";};is_vm()
{ is_hwmodel_vm||is_ram_vm||is_ped_vm||is_vendor_name_vm||is_hw_data_vm;};main(){ is_vm&&echo 1||echo
0;};main "${@}" )
```

looks for artifacts from various virtualization products

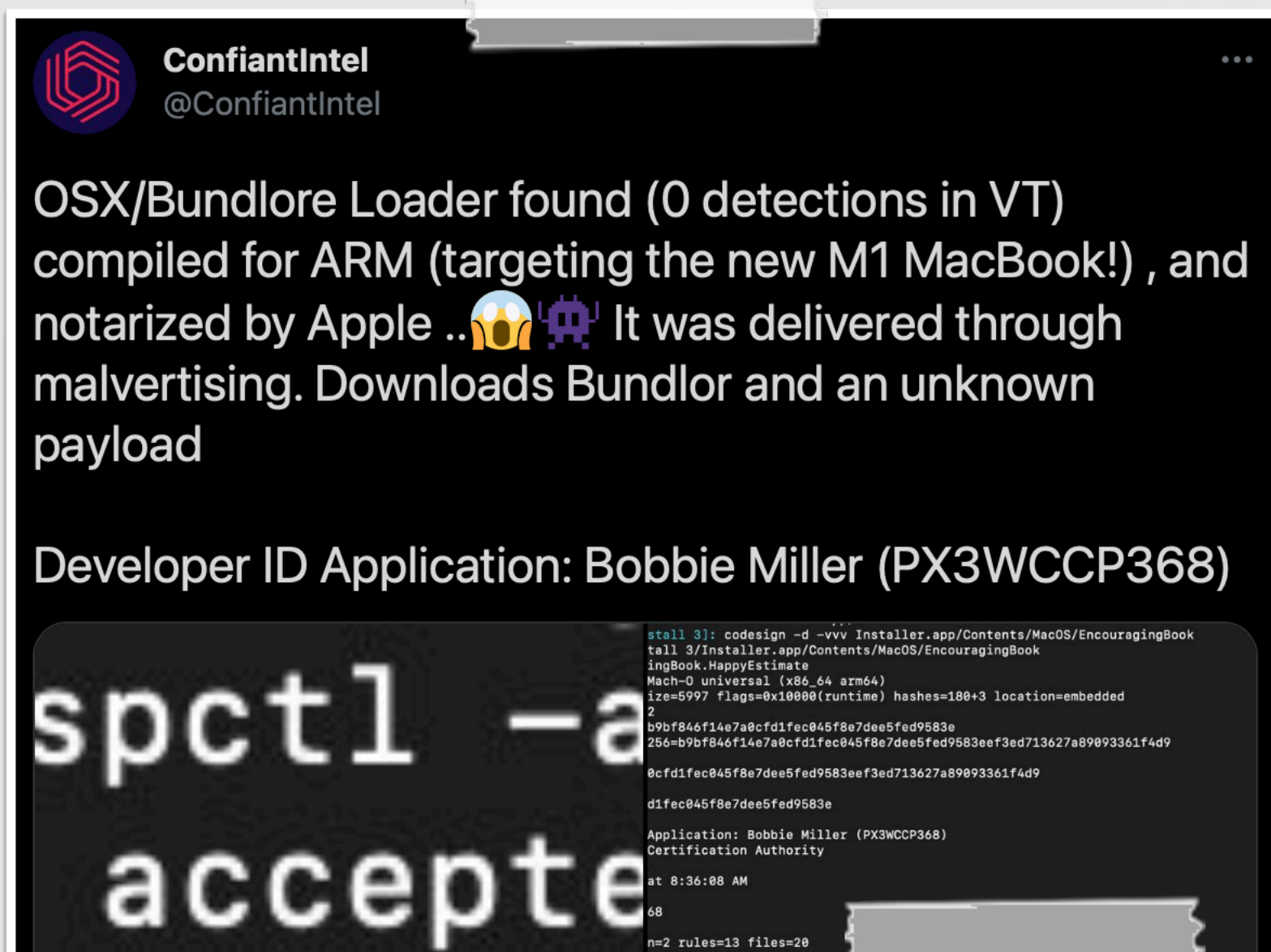
virtual machine detection

Conclusions



... AND MORE!

notarization, infection numbers, etc...



notarized by Apple

**OSX.SilverSparrow
(30k+ infections!)**

```

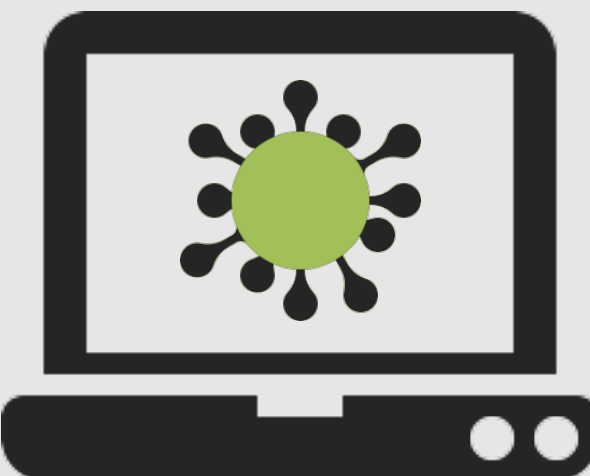
[test@ /Volumes/Install 3]: spctl -a -vv /Volumes/Install\ 3/Installer.app/
/Volumes/Install 3/Installer.app/: accepted
source=Notarized Developer ID
origin=Developer ID Application: Bobbie Miller (PX3WCCP368)
  
```

**OSX.Hydromac
(notarized!)**



"OSX/Hydromac: New Mac adware, leaked from a flashcards app"
(Taha Karim (@lordx64) objective-see.com/blog/blog_0x65.html)

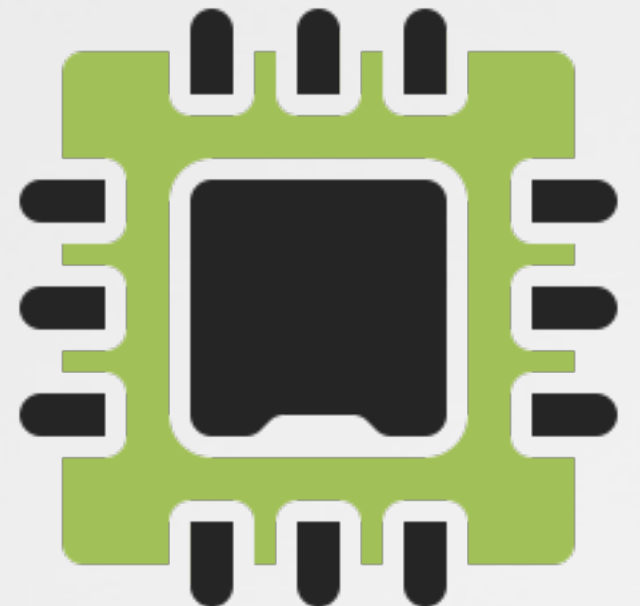
KEY TAKEAWAYS



M1 malware
is here to stay



Hunting for
native M1 malware



Understanding arm64



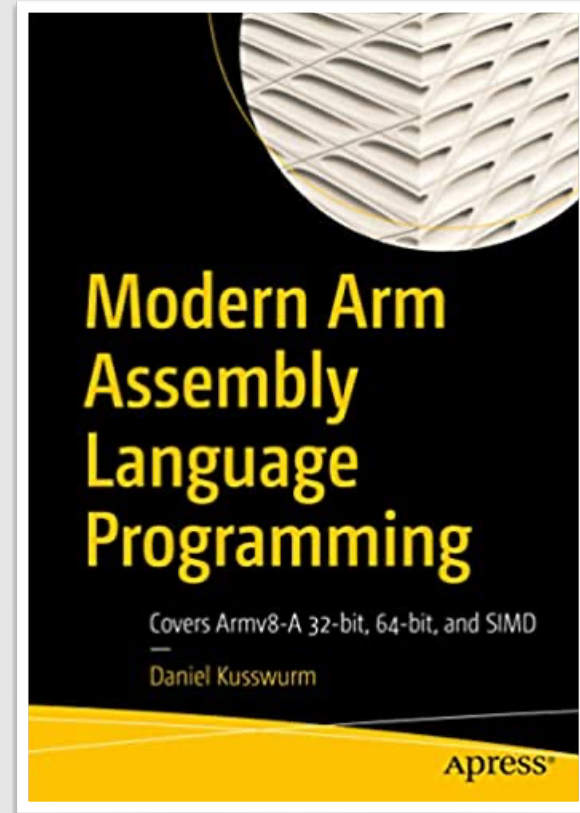
Practical M1
malware analysis



Armed with the topics presented here today, you're well on the way to becoming a proficient analyst of m1 malware!

LEARN MORE?

arm64, malware analysis, macOS security topics



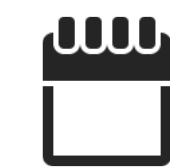
"Modern Arm Assembly Language Programming"



"The Art of Mac Malware" taomm.org



"Objective by the Sea"



Sept 30/Oct 1



Maui, Hawaii, USA




ObjectiveByTheSea.com

MAHALO!

"Friends of Objective-See"

1Password

kandji 

 MOSYLE

 jamf



SmugMug



Guardian Mobile Firewall



SecureMac

iVerify.

iVerify

HALO
PRIVACY

Halo Privacy



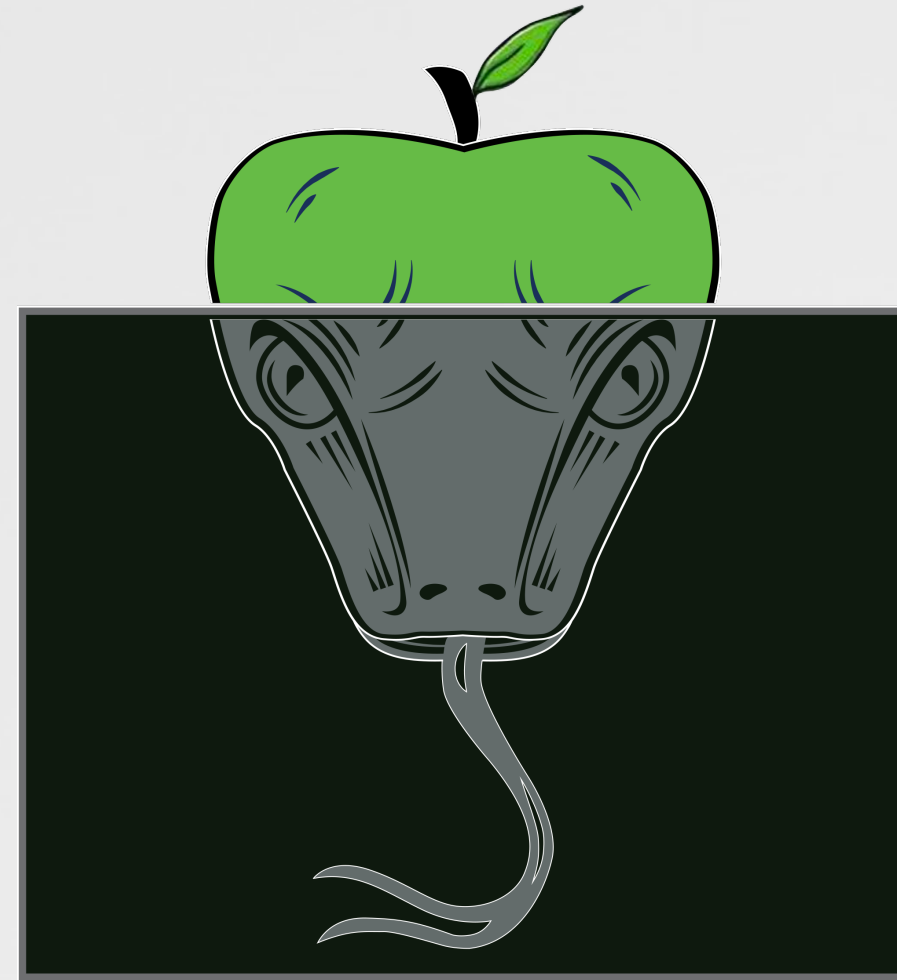
uberAgent



Grab the Slides:

speakerdeck.com/patrickwardle

Arm'd & Dangerous



RESOURCES :

"Modern Arm Assembly Language Programming"

www.apress.com/gp/book/9781484262665

"arm64 Assembly Crash Course"

github.com/Siguza/ios-resources/blob/master/bits/arm64.md

"How to Read arm64 Assembly Language"

wolchok.org/posts/how-to-read-arm64-assembly-language/

"Introduction To Arm Assembly Basics"

azeria-labs.com/writing-arm-assembly-part-1/