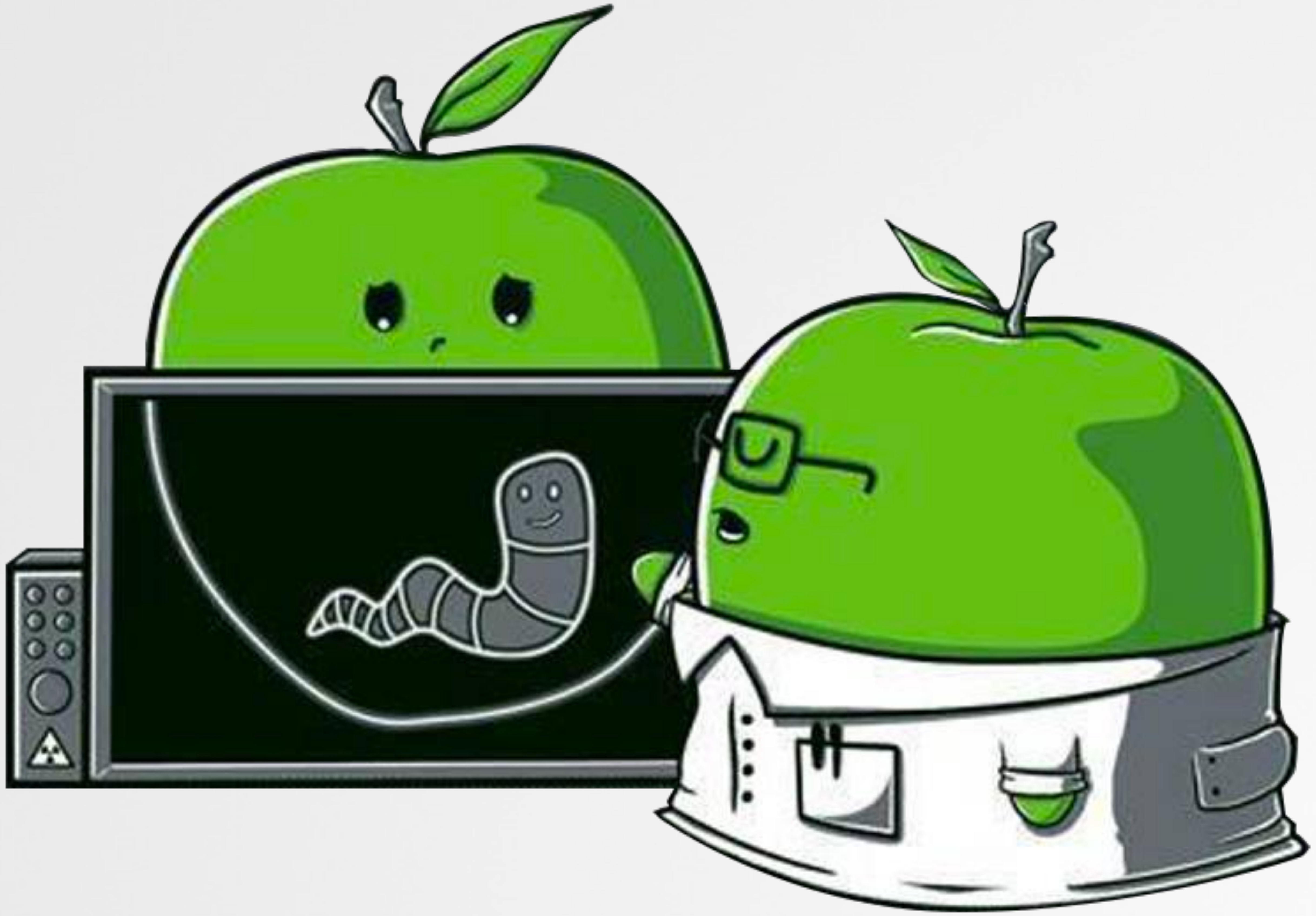
Arm'd & Dangerous

analyzing arm64 malware targeting macOS



@patrickwardle

WHOIS

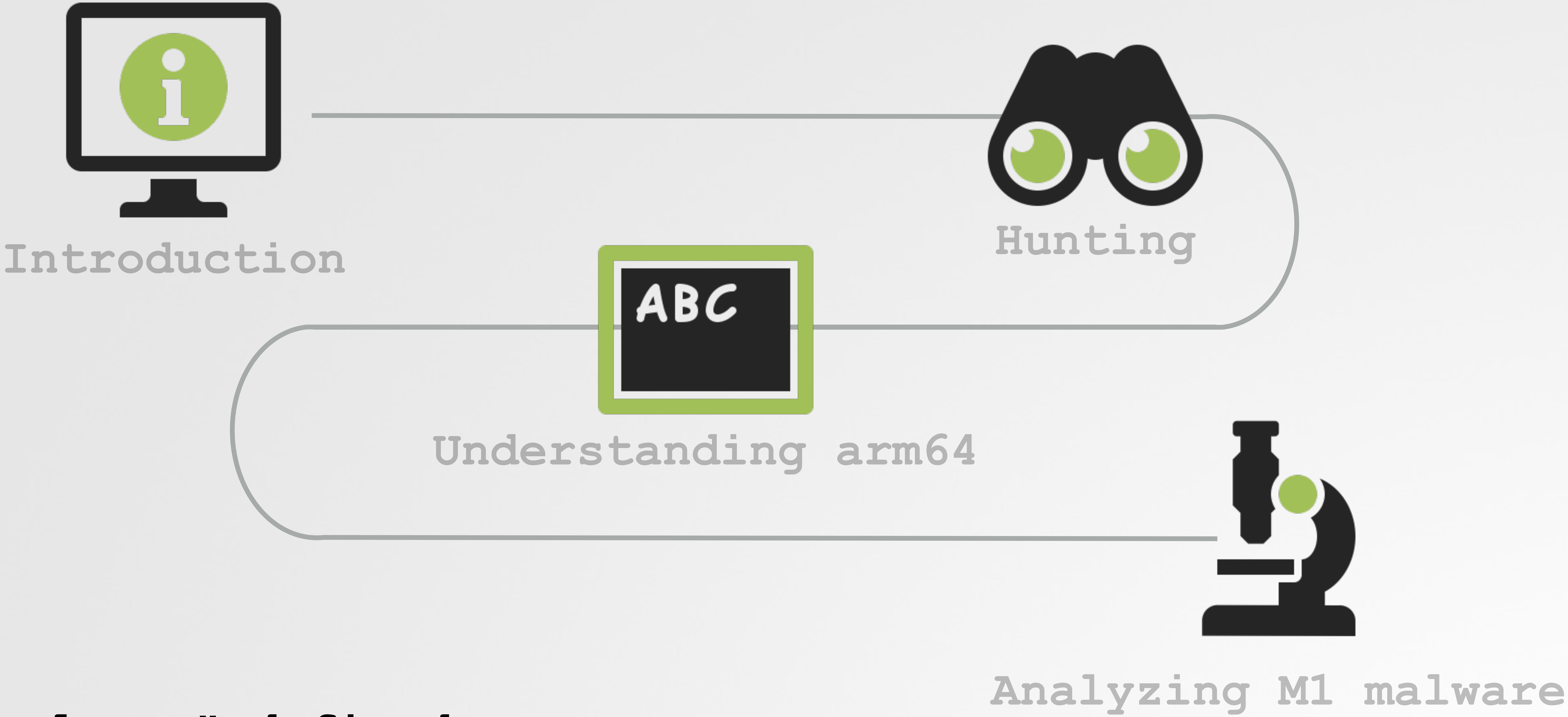
@patrickwardle

Objective-See
tools, blog, & malware collection

#OBTS
"Objective by the Sea"
(macOS security conference)

Book(s):
"The Art of Mac Malware"

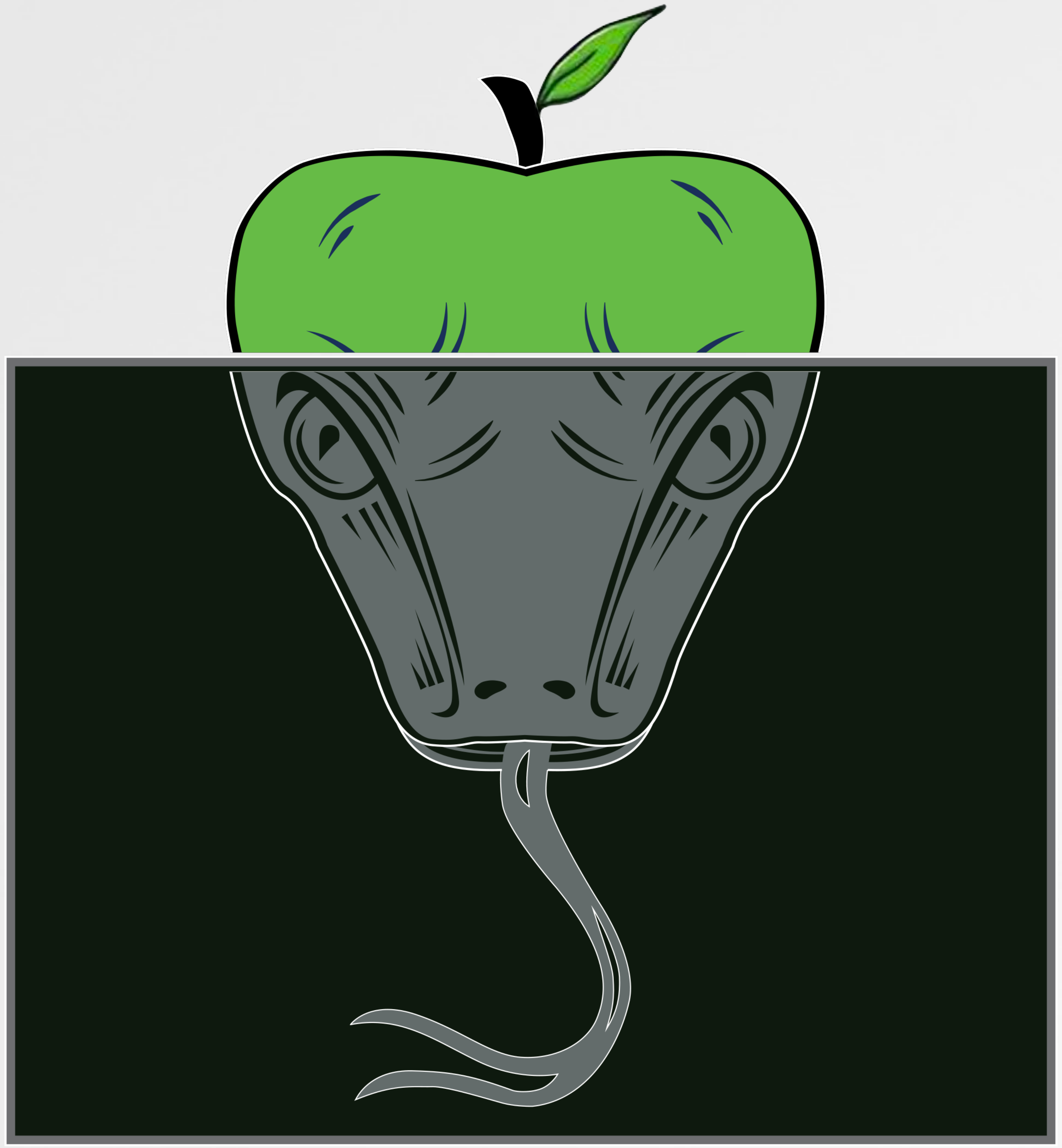
Outline



"M1 malware" defined:

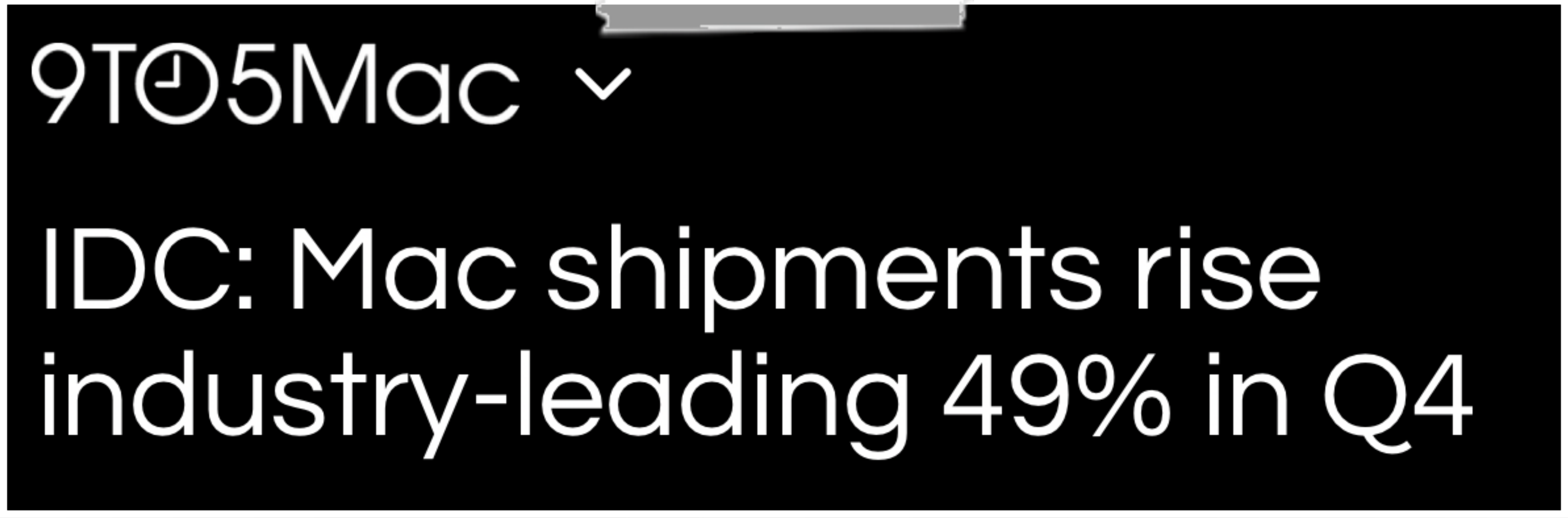
Malware natively compiled to run on Apple Silicon.
More specifically, arm64 malware targeting Macs (macOS).

Introduction



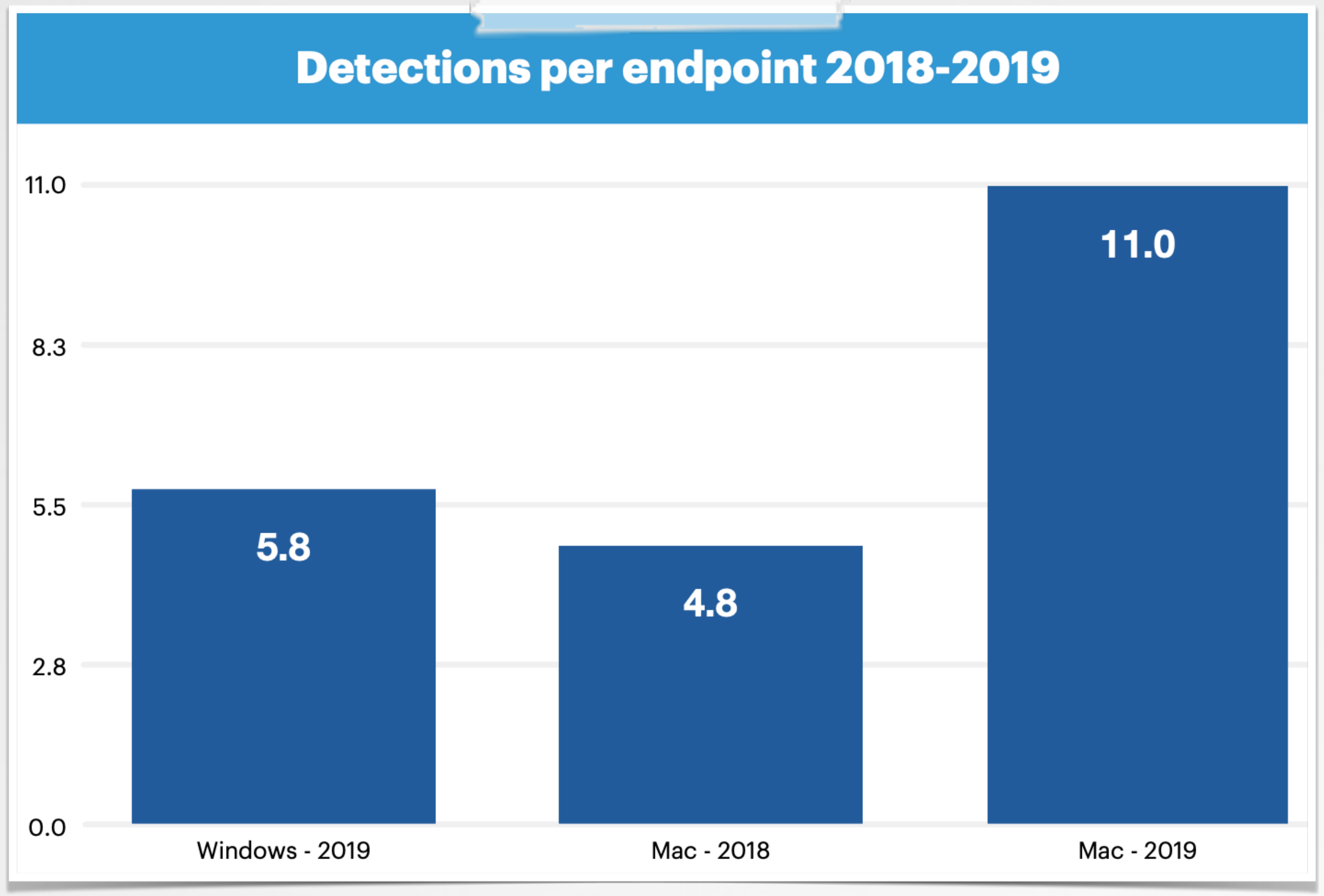
The Growth of Macs

…and unsurprisingly, the growth of macOS malware


9TO5Mac

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 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.

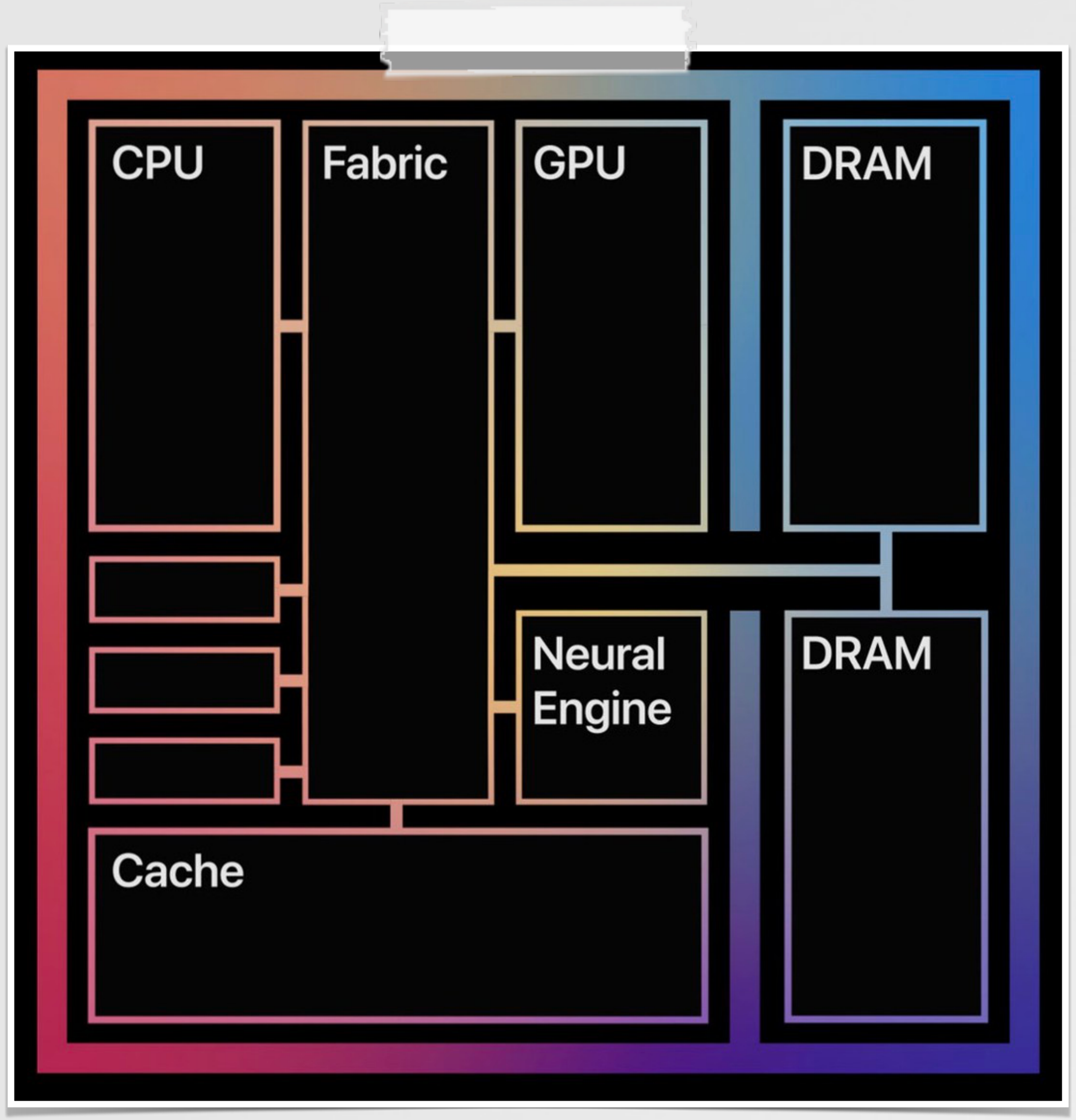
Mac sales are surging thanks to Apple's new M1 processor

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)

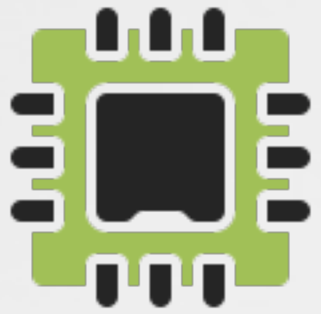
What is M1? (aka "Apple Silicon")

an arm-based system on a chip (SoC)



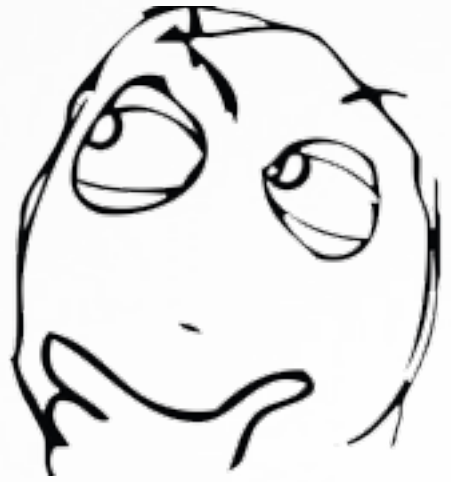
The M1 chip

Multiple technologies combined on a single chip



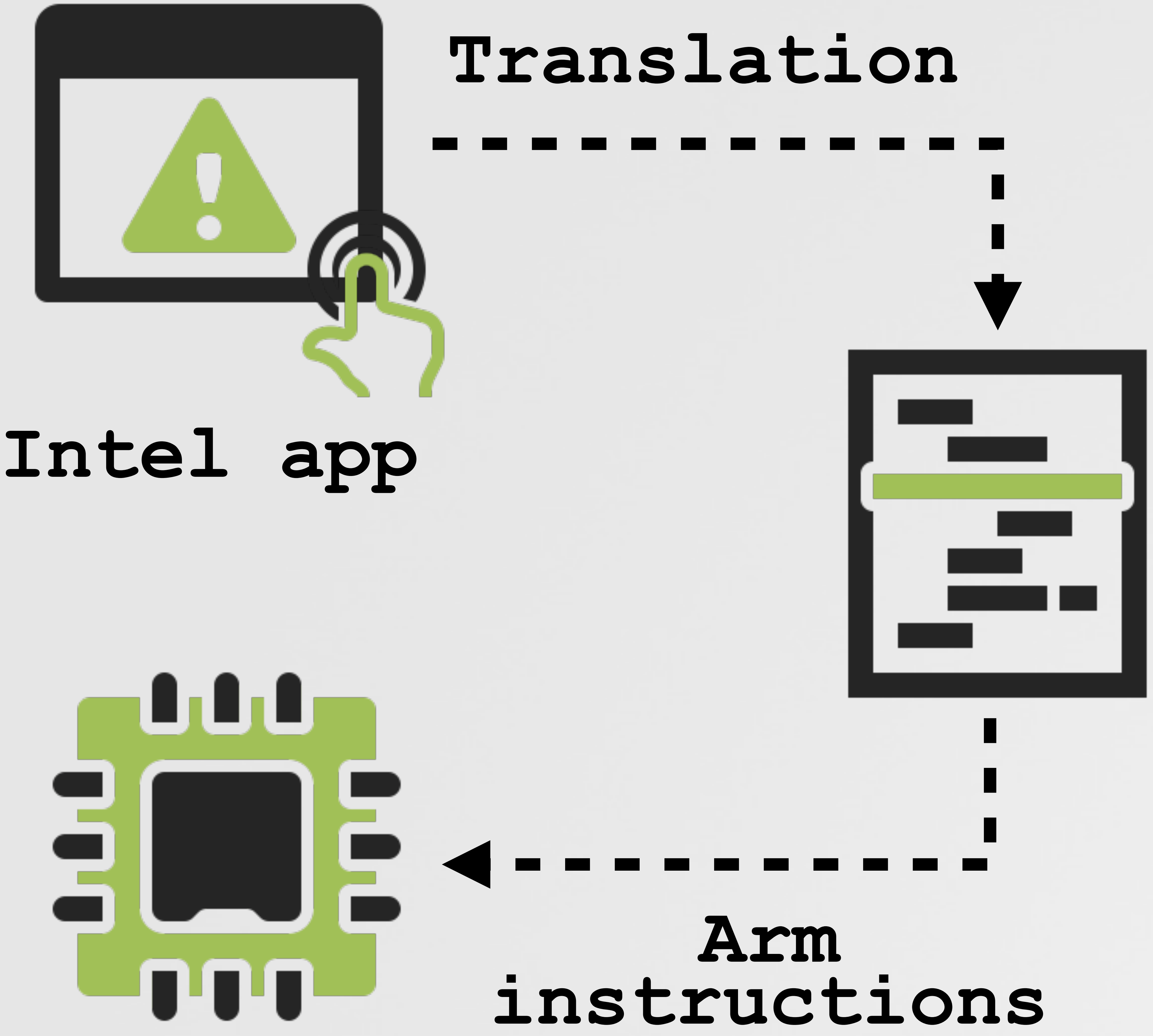
CPU:
arm64 instruction set

...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

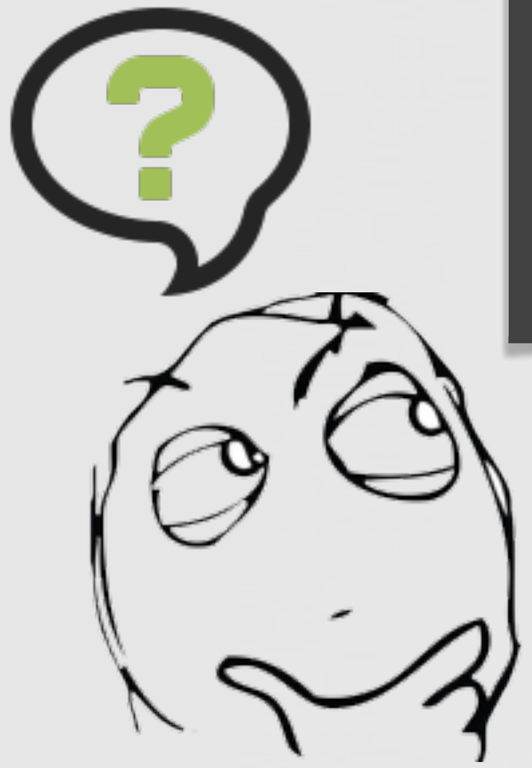
well, several important reasons!

Is (was) inevitable

- → no rosetta crashes
- → native code, faster

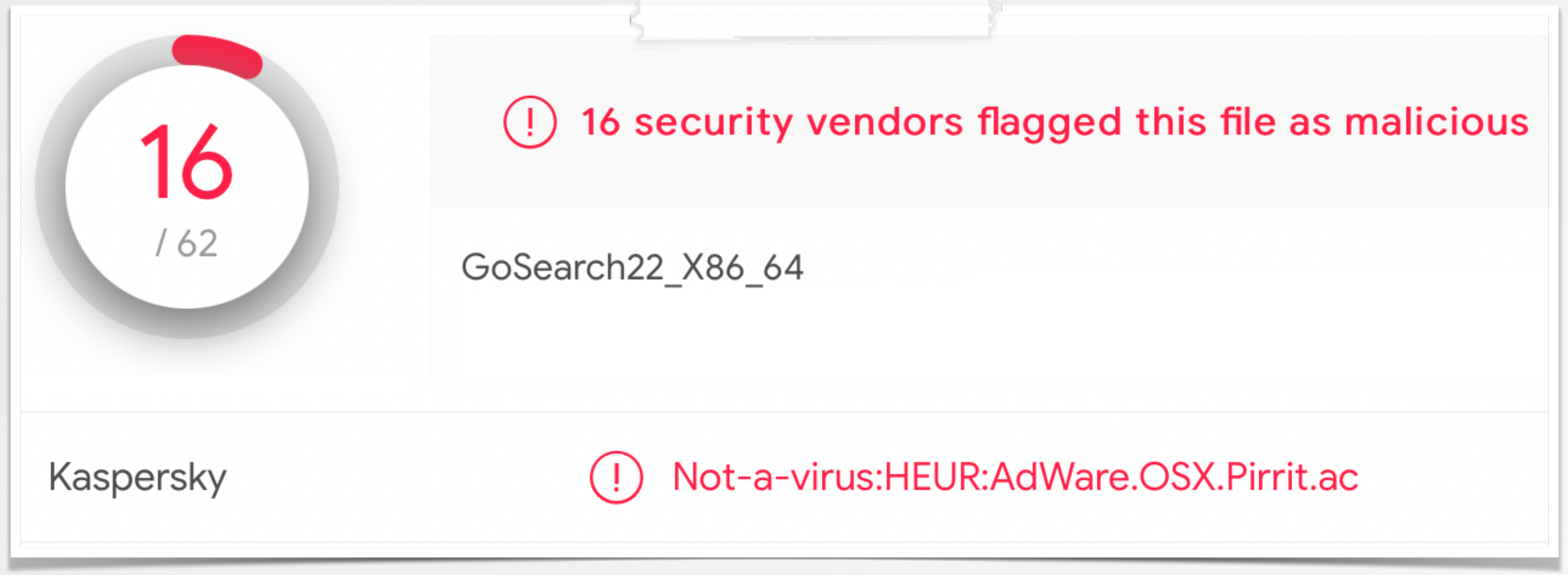
Disassembly is arm64

```
movz    x0, #0x1a
movz    x1, #0x1f
movz    x2, #0x0
movz    x3, #0x0
movz    x16, #0x0
svc     #0x80
```

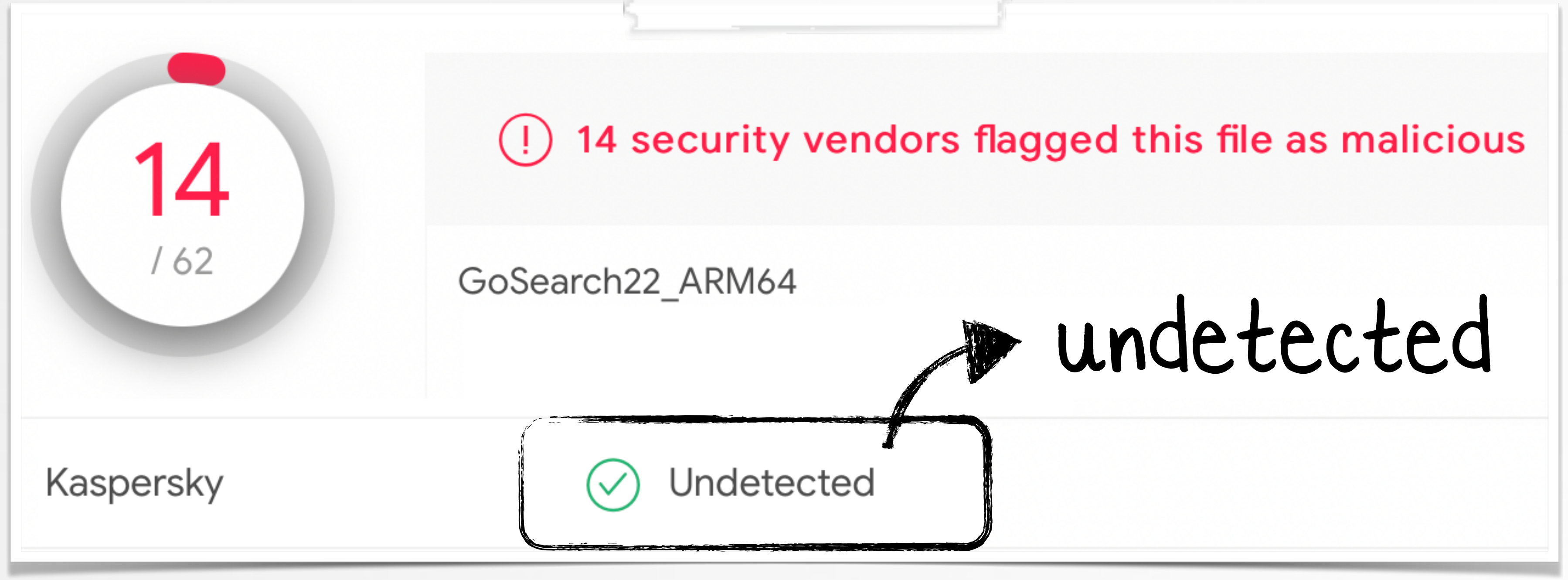


an unfamiliar instructions set!?

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



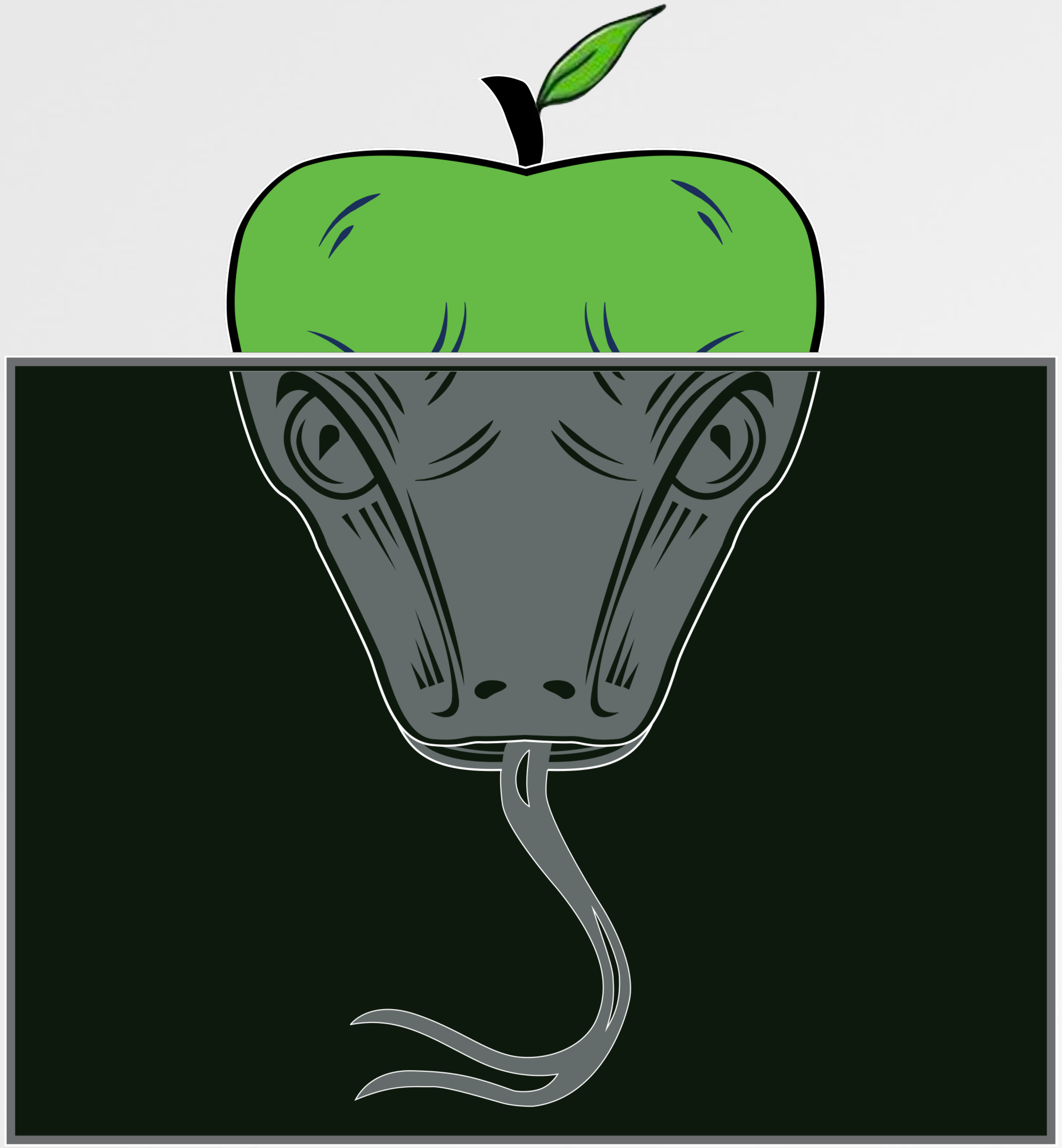
Intel version: detected



Arm version: not detected

same malware

Hunting for M1 malware



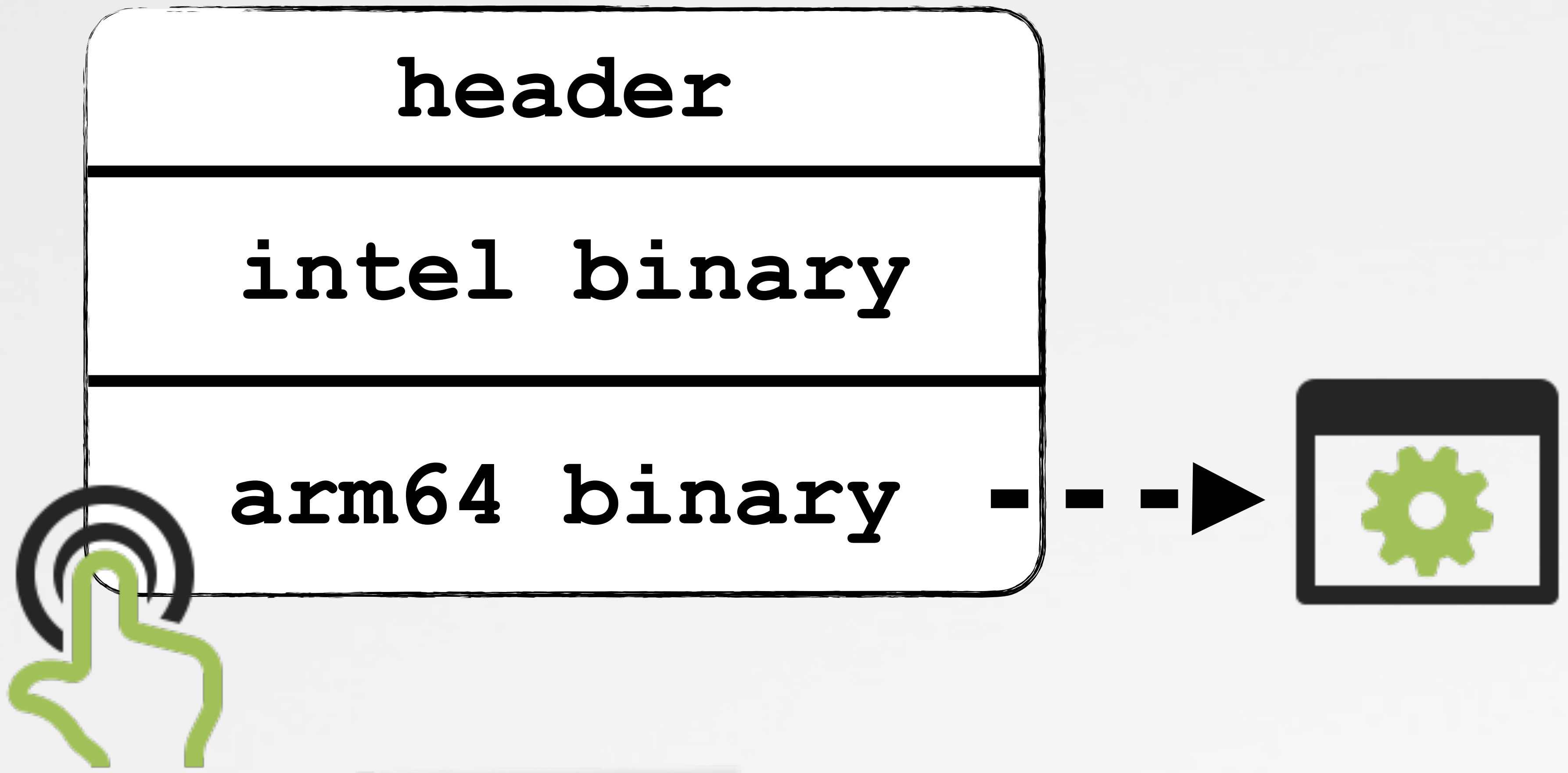
How to Identify M1 code

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)

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)

What's arm64e?

arm64 enhanced
+pointer auth, etc.

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

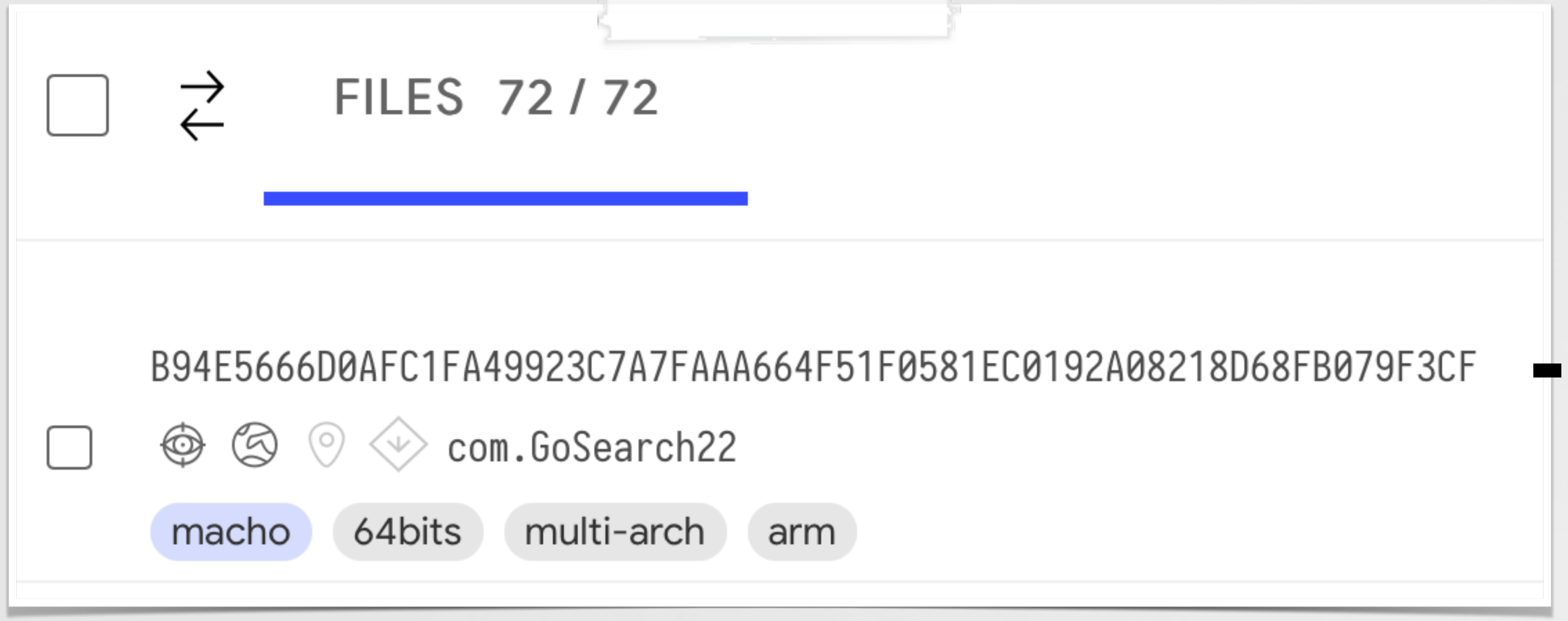
NOT engines:IOS
not an iOS binary

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

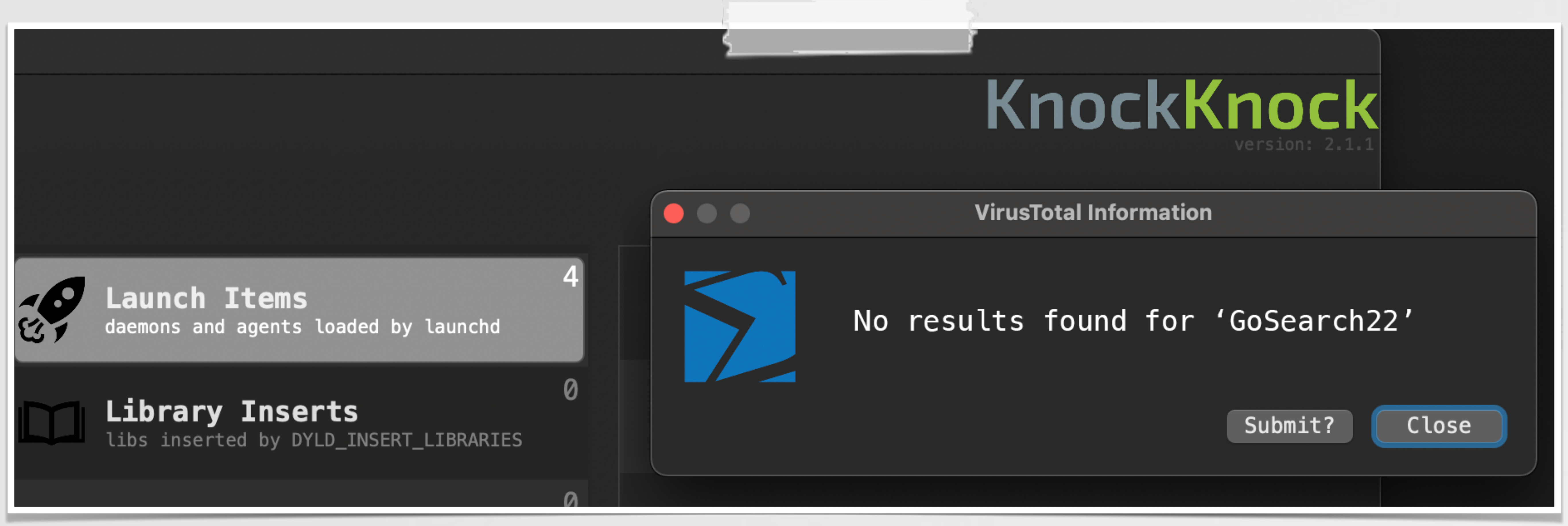
```
% file GoSearch22
Mach-O universal binary with 2 architectures:
 [arm64:Mach-O 64-bit executable arm64]
 [x86_64:Mach-O 64-bit executable x86_64]

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

Universal macOS binary (with arm64)

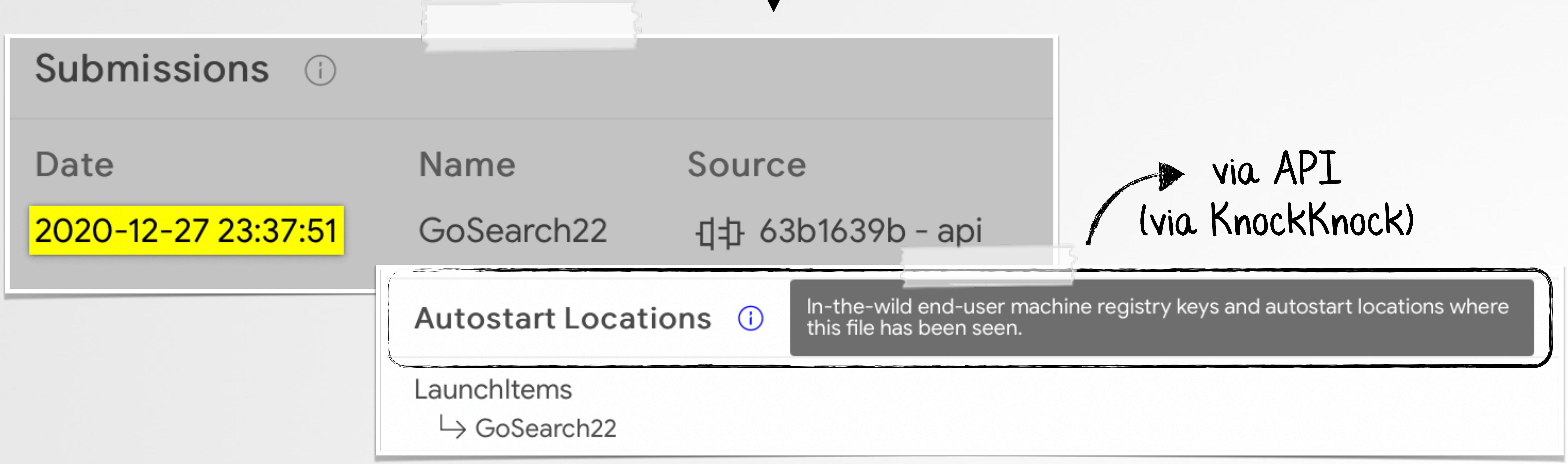
Certificate revoked (by Apple)

How Did it end up on VirusTotal?

…detected and submitted via KnockKnock!

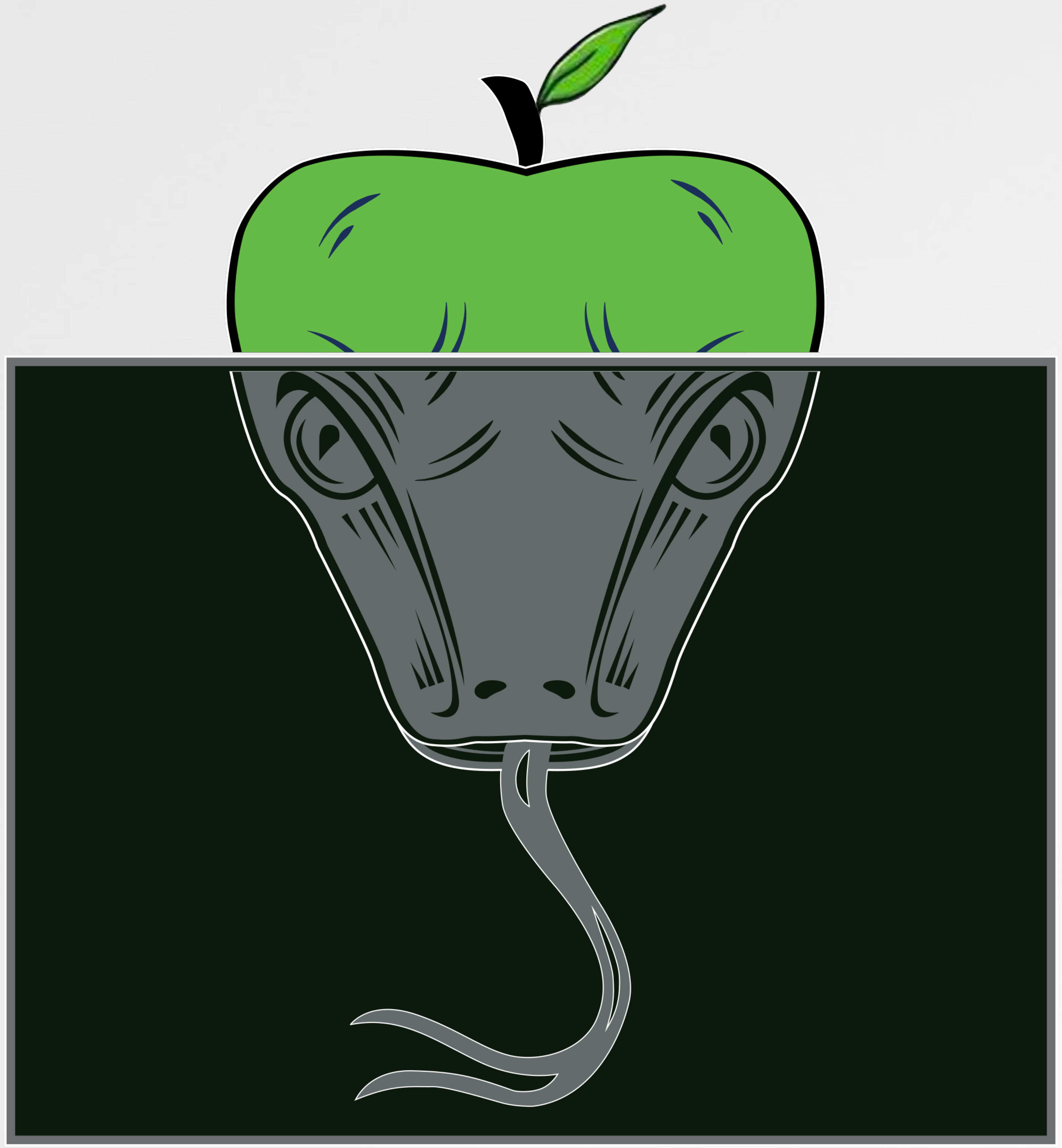


KnockKnock detection
(free: objective-see.com)



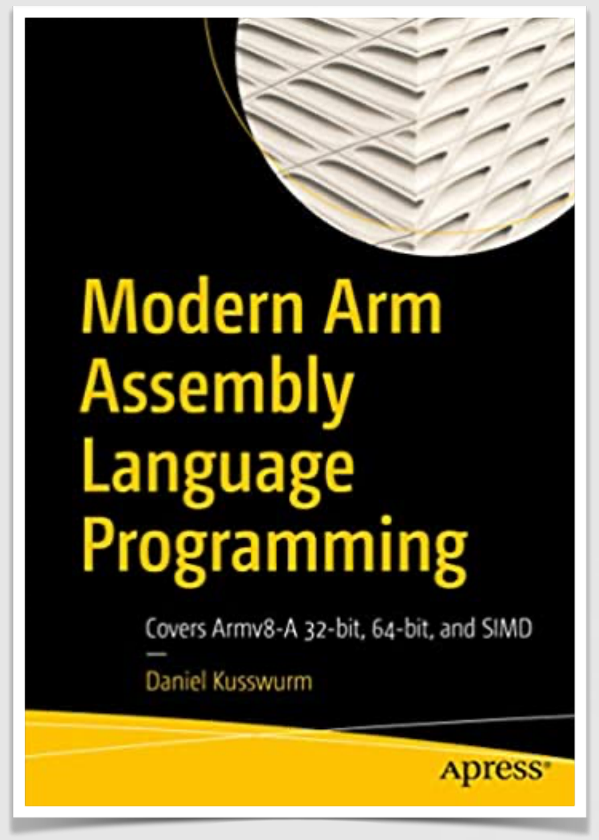
via API
(via KnockKnock)

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/

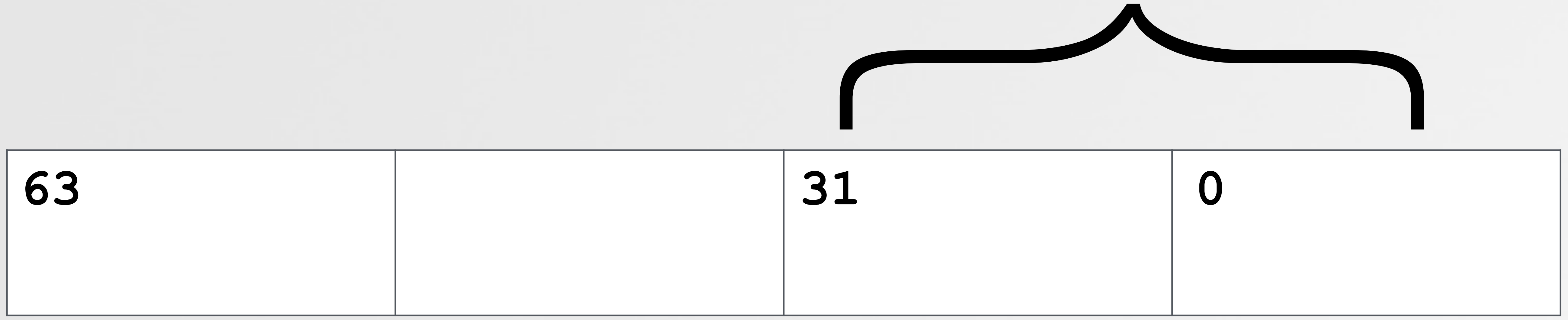
Registers
and their uses

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

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

arm64

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



PSTATE
(processor state)

N	Z	C	V	...

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:

- arg 0 --▶ x0
- arg 1 --▶ x1
- ...
- arg 7 --▶ x7

Frame pointer

- --▶ x29 (fp)

Return address:

- --▶ x30 (lr)

Return value:
(64/128 bits)

- --▶ x0
- --▶ x1

INSTRUCTIONS

instruct the cpu what to do

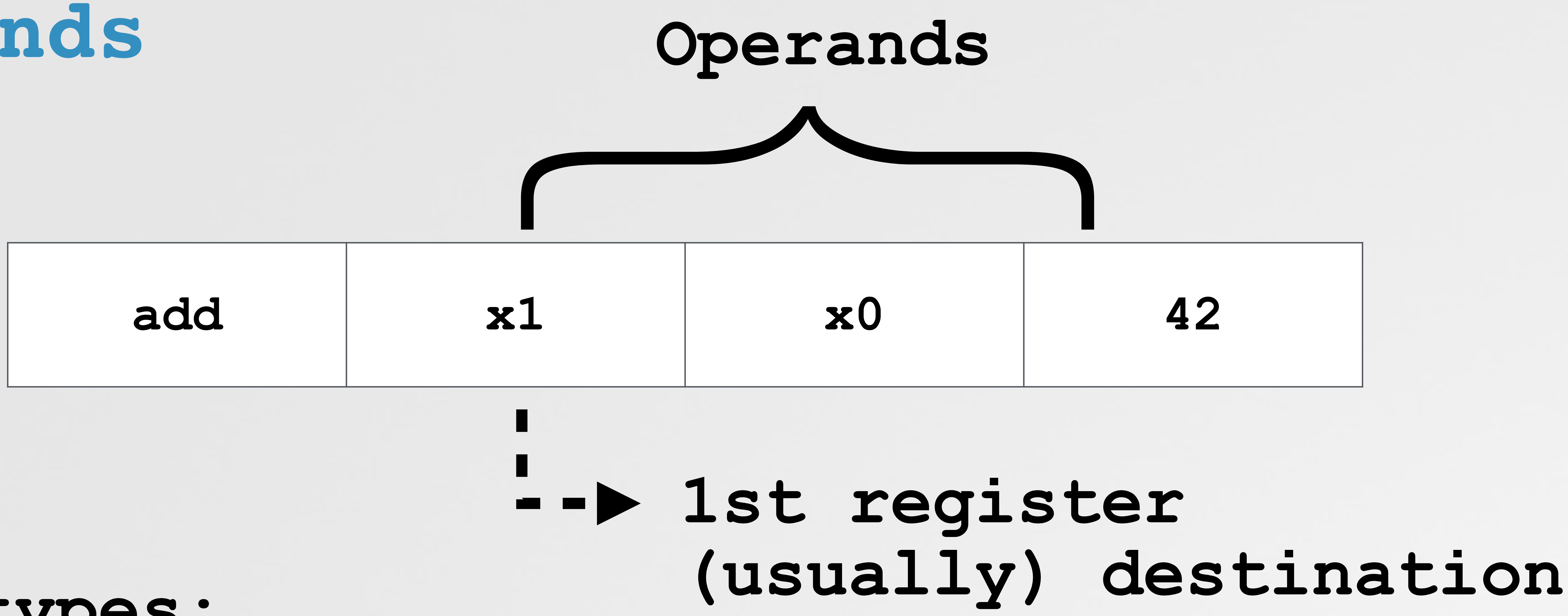
Instructions: map to a specific sequence of bytes that instructs the CPU to perform an operation.

in C: x1 = x0 + 42;

add	x1	x0	42

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

INSTRUCTIONS
the operands



Operand types:

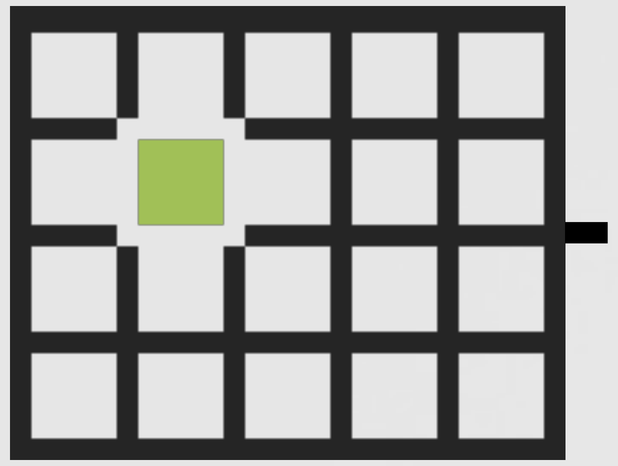
Immediate:
a constant value (e.g. 42)

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

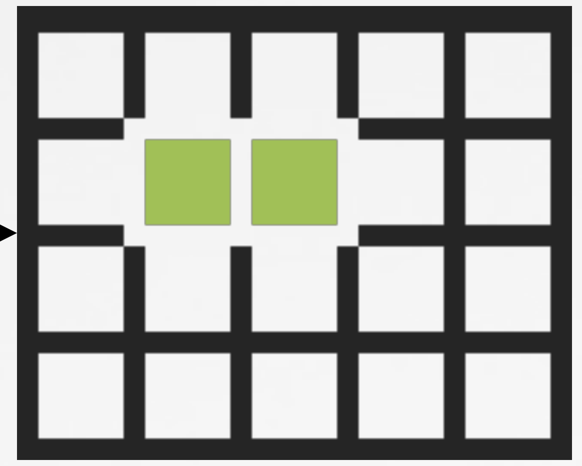
Memory:
a cpu register, that points to a value in memory

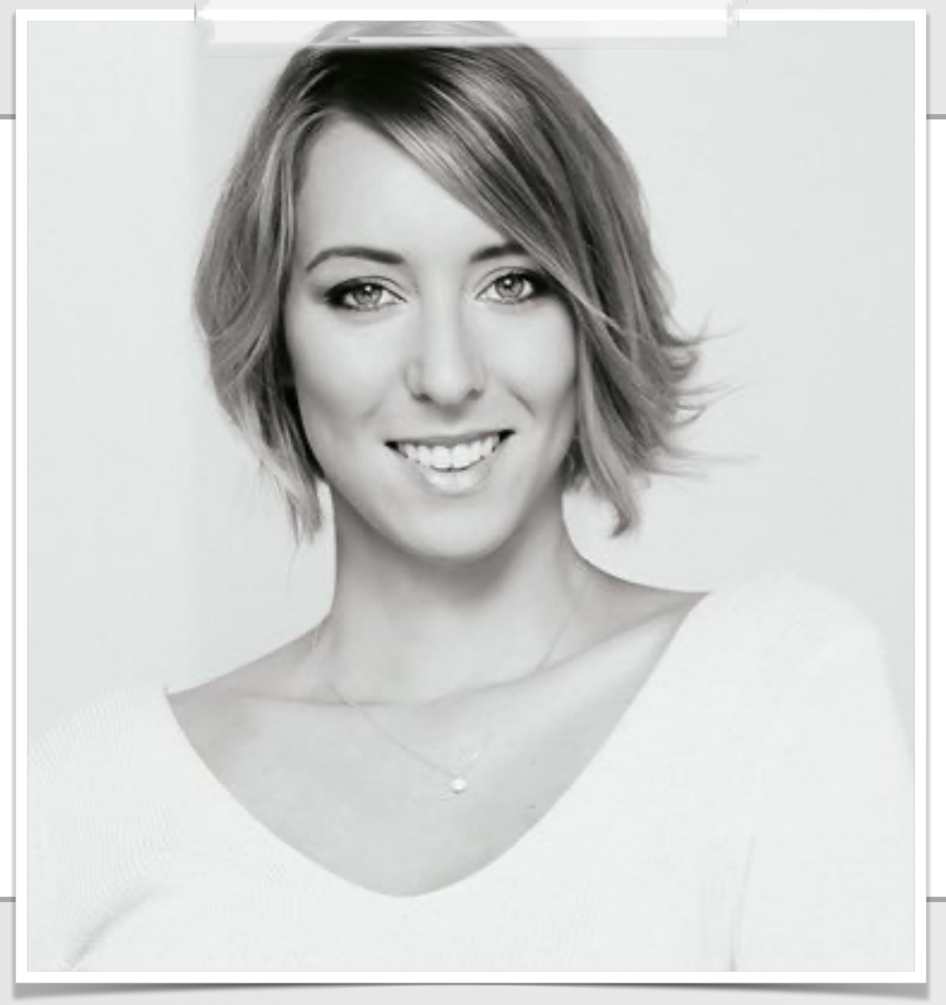
Memory Access Model

arm's model is a "load & store"



1. Load (into register)
2. Perform any operation(s)
3. Store (into memory)





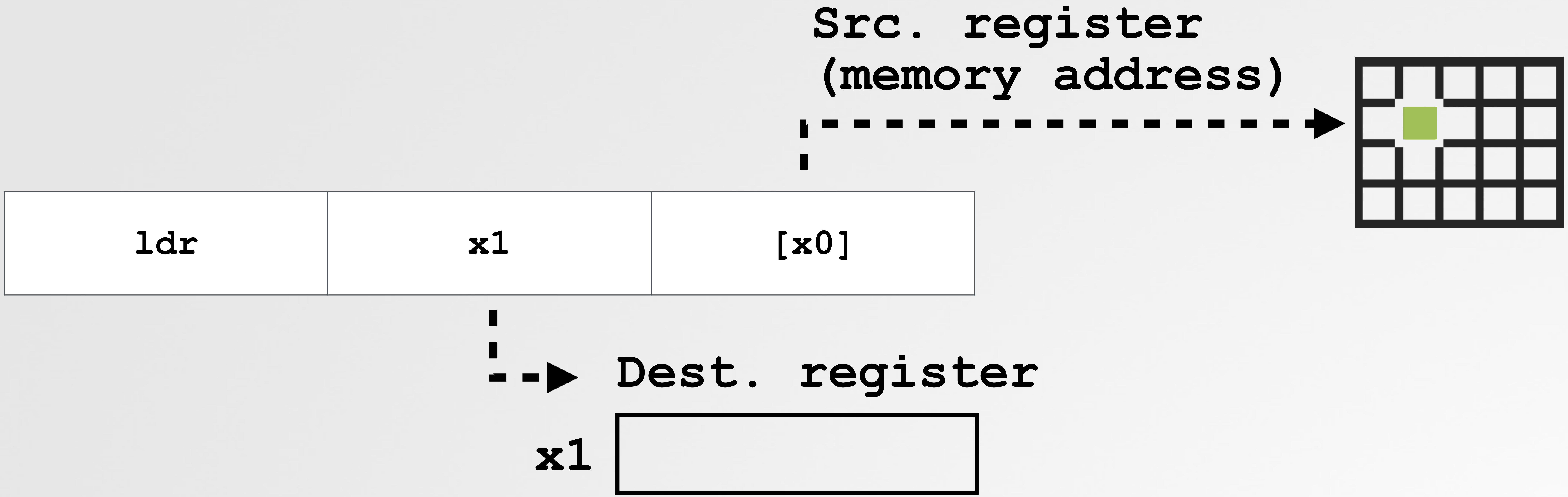
"ARM uses a load-store model for memory access which means that only load/store (LDR and STR) instructions can access memory.

…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)

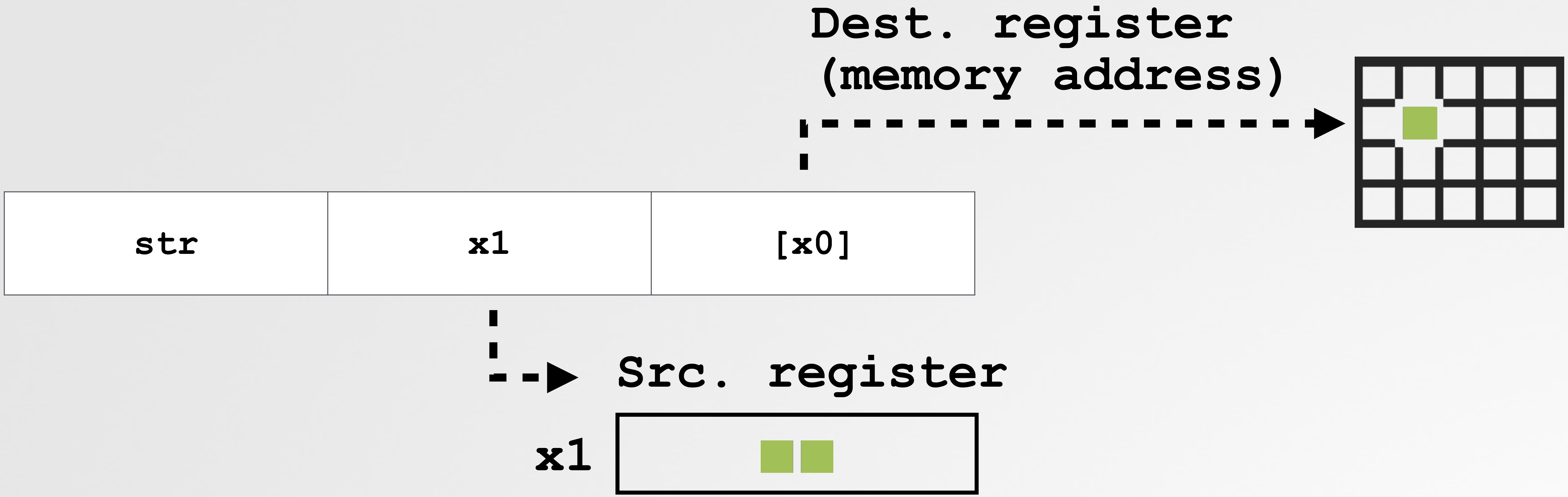


Analogous statement (in C):

```
x1 = *x0;
```

Memory Access Model

store via the str instruction (+ variants)

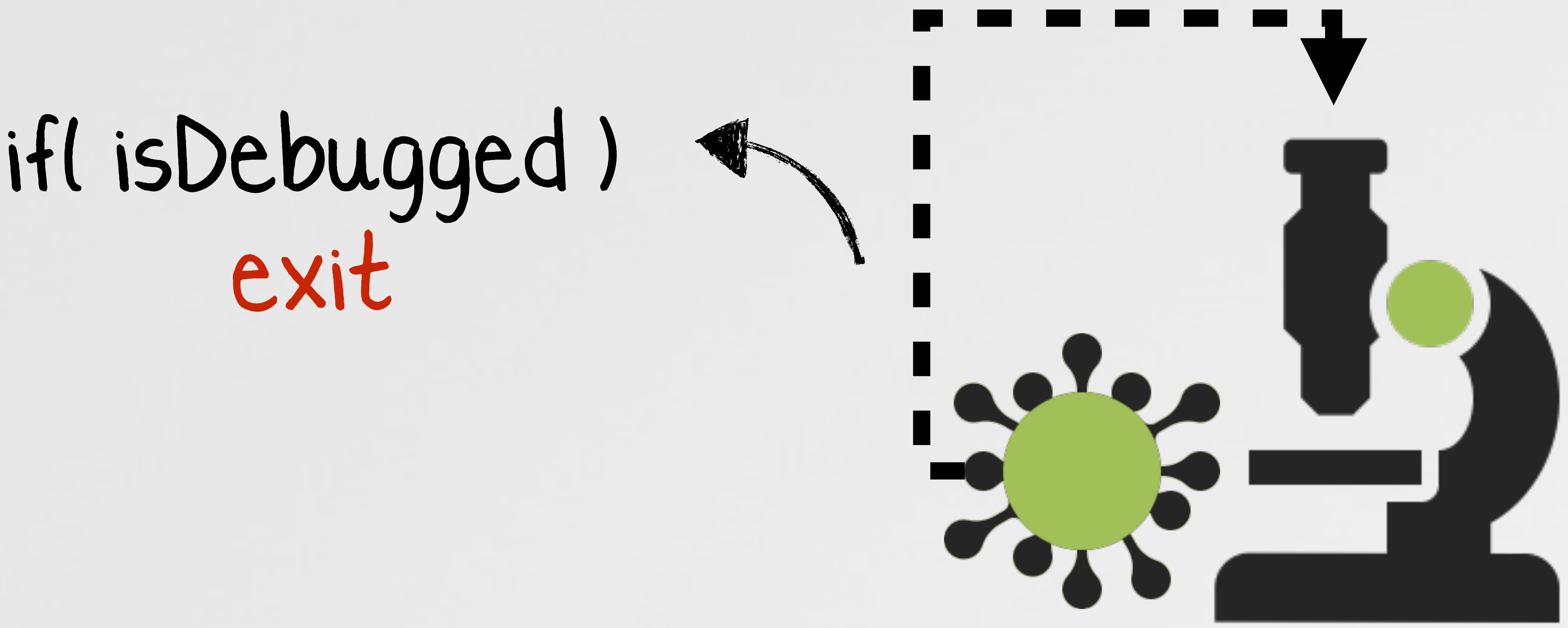


Analogous statement (in C):

```
*x0 = x1;
```

Conditions

set via cmp, etc…



cmp	x0	42

(discarded) subtract
updates PSTATE flags

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

PSTATE
(processor state)

N	Z	C	V	...

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

```
bl      amBeingDebugged
cmp     w0, #1
b.ne    continue
movn    w0, #0x0
bl      exit

continue:
…
```

exit if debugged

b.ne	label

Branch (jump), if Z not set

Branches

alter control flow of a program

1. | b/br | imm/register |
|---|---|

Branch (unconditionally)

2. | b.cond | imm |
|---|---|

Branch (if condition met)

```
bl      amBeingDebugged
cmp     w0, #1
b.ne    continue
movn    w0, #0x0
bl      exit

continue:
…
```

3. | bl/blr | imm/register |
|---|---|

e.g. function call

Branch (store address of next instruction in x30 (lr))

```
bl      amBeingDebugged
cmp     w0, #1
...
```

ret

Branch back to x30 (lr)

&next instruction (cmp w0, #1)

x30 (lr)

Reversing "Hello, World!"

macOS arm64 version

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

(Apple's) "Hello, World!"

```
main:
 sub    sp, sp, #0x30
 stp    x29, x30, [sp, #0x20]
 add    x29, sp, #0x20
 movz   w8, #0x0
 stur   wzr, [x29, #-0x4]
 stur   w0, [x29, #-0x8]
 str    x1, [sp, #0x10]
 str    w8, [sp, #0xc]
 bl     objc_autoreleasePoolPush
 adrp   x9, #0x0000000100004000
 add    x9, x9, #0x8 ; @"Hello,  World!"
 str    x0, [sp]
 mov    x0, x9
 bl     NSLog
 ldr    x0, [sp]
 bl     objc_autoreleasePoolPop
 ldr    w0, [sp, #0xc]
 ldp    x29, x30, [sp, #0x20]
 add    sp, sp, #0x30
 ret
```

"Hello, World!" disassembled

Note, @autoreleasepool:

1. objc_autoreleasePoolPush

+

2. objc_autoreleasePoolPop

REVERSING "HELLO, WORLD!"

function prologue

```
main:
sub     sp, sp, #0x30
stp     x29, x30, [sp, #0x20]
add     x29, sp, #0x20
movz    w8, #0x0
stur    wzr, [x29, #-0x4]
stur    w0, [x29, #-0x8]
str     x1, [sp, #0x10]
str     w8, [sp, #0xc]
bl      objc_autoreleasePoolPush
adrp    x9, #0x0000000100004000
add     x9, x9, #0x8 ; @"Hello, World!"
str     x0, [sp]
mov     x0, x9
bl      NSLog
ldr     x0, [sp]
bl      objc_autoreleasePoolPop
ldr     w0, [sp, #0xc]
ldp     x29, x30, [sp, #0x20]
add     sp, sp, #0x30
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 (before)	0x30	
	0x28	x30
x29	0x20	x29
	...	
sp (after)	0x00	

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

Reversing "Hello, World!"

invoking objc_autoreleasePoolPush

```
main:
sub     sp, sp, #0x30
stp     x29, x30, [sp, #0x20]
add     x29, sp, #0x20
movz    w8, #0x0
stur    wzr, [x29, #-0x4]
stur    w0, [x29, #-0x8]
str     x1, [sp, #0x10]
str     w8, [sp, #0xc]
bl      objc_autoreleasePoolPush
adrp    x9, #0x0000000100004000
add     x9, x9, #0x8 ; @"Hello, World!"
str     x0, [sp]
mov     x0, x9
bl      NSLog
ldr     x0, [sp]
bl      objc_autoreleasePoolPop
ldr     w0, [sp, #0xc]
ldp     x29, x30, [sp, #0x20]
add     sp, sp, #0x30
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

```
main:
sub     sp, sp, #0x30
stp     x29, x30, [sp, #0x20]
add     x29, sp, #0x20
movz    w8, #0x0
stur    wzr, [x29, #-0x4]
stur    w0, [x29, #-0x8]
str     x1, [sp, #0x10]
str     w8, [sp, #0xc]
bl      objc_autoreleasePoolPush
adrp    x9, #0x0000000100004000
add     x9, x9, #0x8 ; @"Hello, World!"
str     x0, [sp]
mov     x0, x9
bl      NSLog
ldr     x0, [sp]
bl      objc_autoreleasePoolPop
ldr     w0, [sp, #0xc]
ldp     x29, x30, [sp, #0x20]
add     sp, sp, #0x30
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

```
main:
sub     sp, sp, #0x30
stp     x29, x30, [sp, #0x20]
add     x29, sp, #0x20
movz    w8, #0x0
stur    wzr, [x29, #-0x4]
stur    w0, [x29, #-0x8]
str     x1, [sp, #0x10]
str     w8, [sp, #0xc]
bl      objc_autoreleasePoolPush
adrp    x9, #0x0000000100004000
add     x9, x9, #0x8 ; @"Hello, World!"
str     x0, [sp]
mov     x0, x9
bl      NSLog
ldr     x0, [sp]
bl      objc_autoreleasePoolPop
ldr     w0, [sp, #0xc]
ldp     x29, x30, [sp, #0x20]
add     sp, sp, #0x30
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

```
main:
sub     sp, sp, #0x30
stp     x29, x30, [sp, #0x20]
add     x29, sp, #0x20
movz    w8, #0x0
stur    wzr, [x29, #-0x4]
stur    w0, [x29, #-0x8]
str     x1, [sp, #0x10]
str     w8, [sp, #0xc]
bl      objc_autoreleasePoolPush
adrp    x9, #0x0000000100004000
add     x9, x9, #0x8 ; @"Hello, World!"
str     x0, [sp]
mov     x0, x9
bl      NSLog
ldr     x0, [sp]
bl      objc_autoreleasePoolPop
ldr     w0, [sp, #0xc]
ldp     x29, x30, [sp, #0x20]
add     sp, sp, #0x30
ret
```

Initialize return value (previously set to zero)

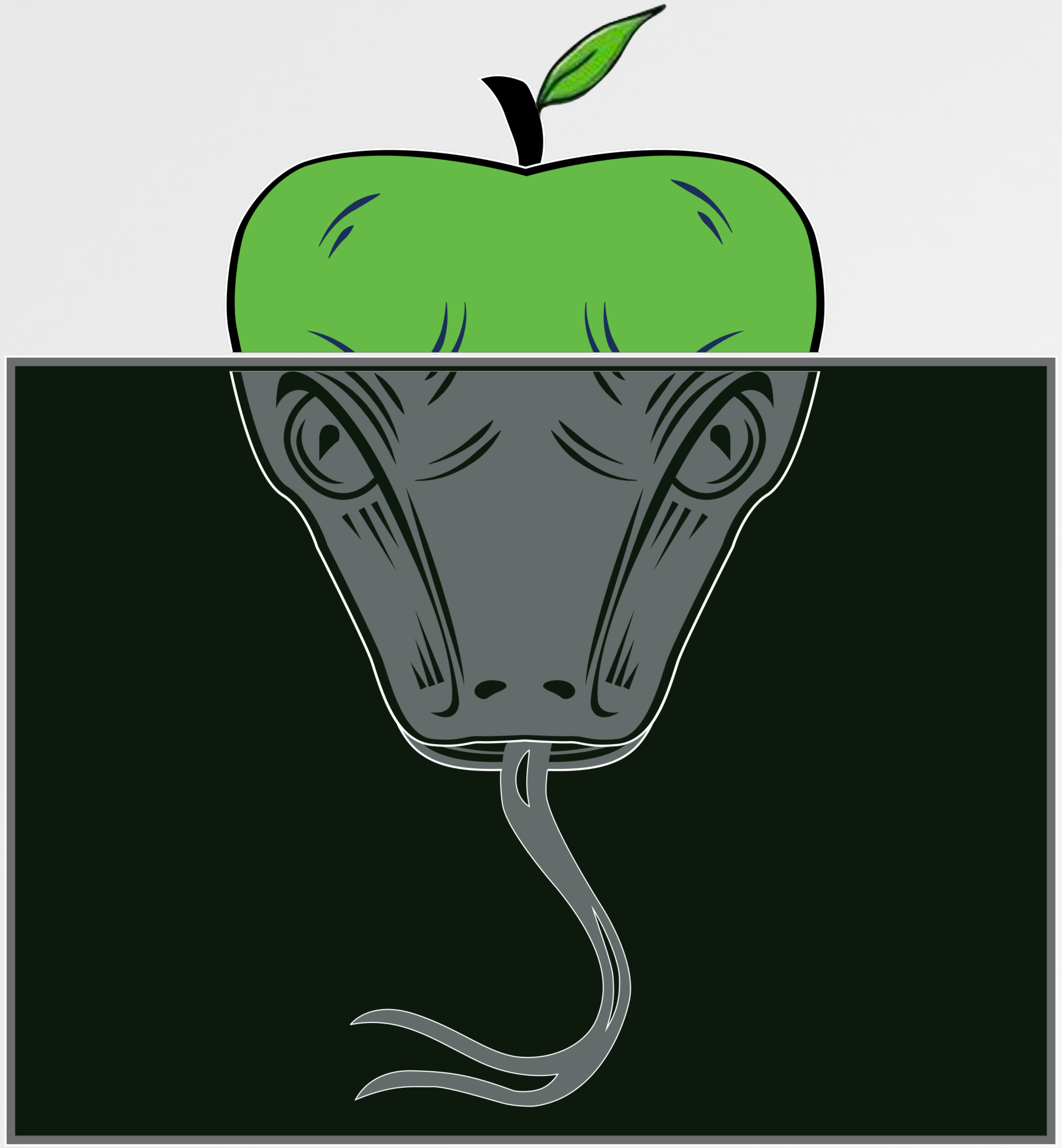
Restore x29/x30 registers

(re)Adjust stack

Return to caller (lr/x30)

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!

```
main:
 sub    sp, sp, #0x30
 stp    x29, x30, [sp, #0x20]
 add    x29, sp, #0x20
 movz   w8, #0x0
 stur   wzr, [x29, #-0x4]
 stur   w0, [x29, #-0x8]
 str    x1, [sp, #0x10]
 str    w8, [sp, #0xc]
 bl     objc_autoreleasePoolPush
 adrp   x9, #0x0000000100004000
 add    x9, x9, #0x8 ; @"Hello,  World!"
 str    x0, [sp]
 mov    x0, x9
 bl     NSLog
 ldr    x0, [sp]
 bl     objc_autoreleasePoolPop
 ldr    w0, [sp, #0xc]
 ldp    x29, x30, [sp, #0x20]
 add    sp, sp, #0x30
 ret
```

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

"Hello, World!" decompiled

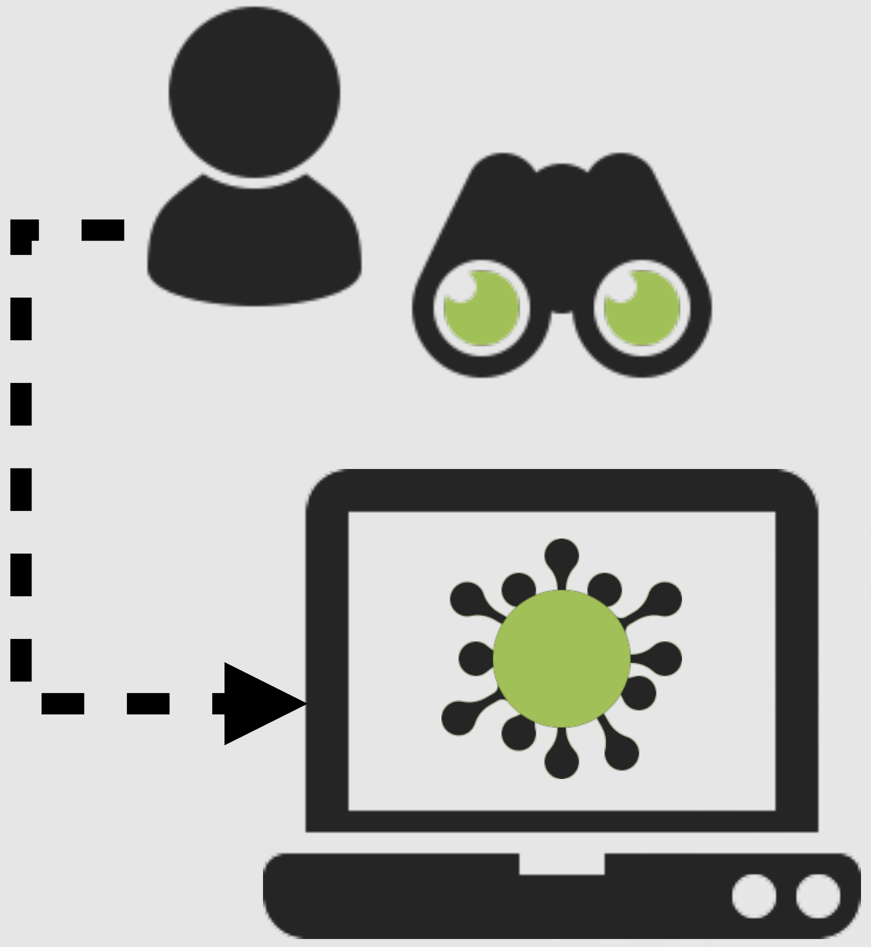
Dynamic Analysis Tools:

- 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



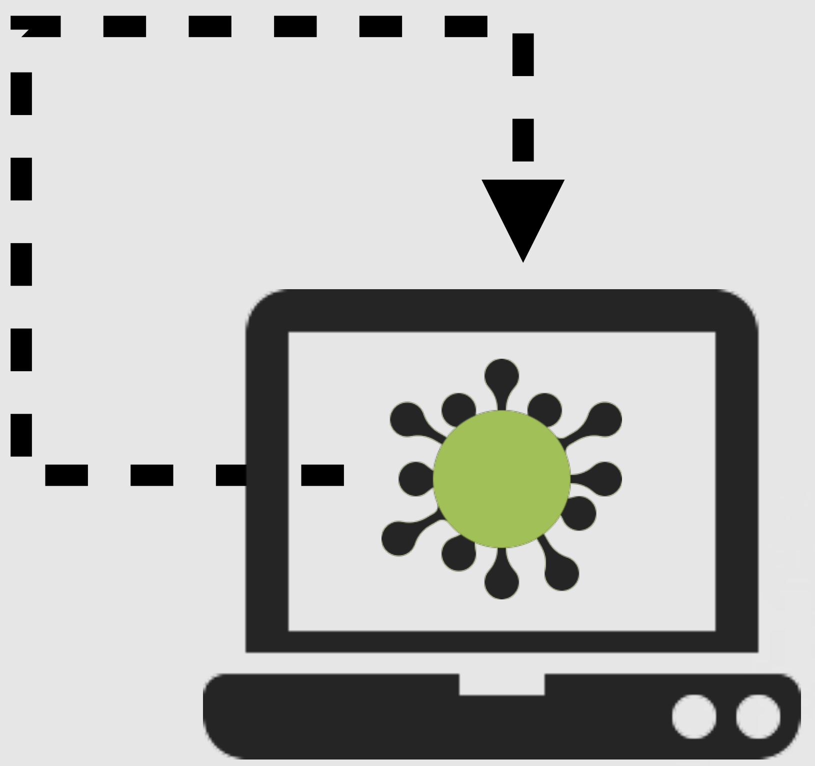

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Users/user/Library/LaunchAgents/mdworker.plist",
    "process" : {
      "uid" : 501,
      "arguments" : [
        "/bin/sh",
        "-c",
        "/Users/user/Desktop/eTrader.app/Contents/Utils/mdworker"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

simply terminates :(

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

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

GoSearch22 vs. debugger

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

GoSearch22

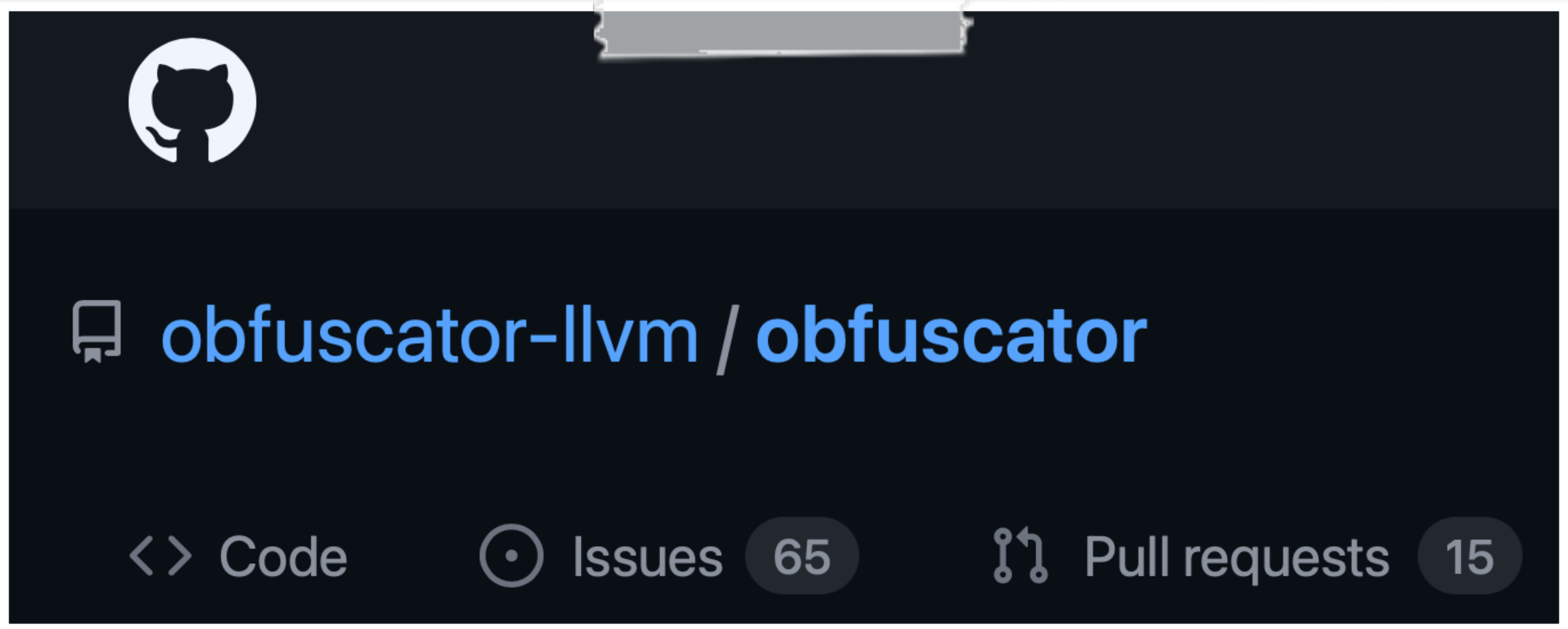
…also contains static analysis obfuscations

```
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

```
*(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;
```

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

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

45 (02xd)
ENOTSUP (from PT_DENY_ATTACH)

GoSearch22 vs. debugger

```
movz    x0, #0x1a
movz    x1, #0x1f
movz    x2, #0x0
movz    x3, #0x0
...
svc     #0x80
```

Args:

0x1a: SYS_ptrace
0x1f: PT_DENY_ATTACH
0x0
0x0

svc → Supervisor (system) call

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

```
0x00000001000541f4 movz   x3, #0x0
0x00000001000541f8 movz   x16, #0x0
0x00000001000541fc svc    #0x80

0x0000000100054200 movz   w11, #0x6b8f
```

simply skip over
ptrace system call :)

```
% lldb GoSearch22.app

(lldb) b 0x00000001000541fc
Breakpoint 1: address = 0x00000001000541fc

(lldb) Process 1486 stopped
* thread #1, queue = 'com.apple.main-thread'
  stop reason = breakpoint 1.1:
->  0x00000001000541fc svc    #0x80

(lldb) reg write $pc 0x100054200
```

modify PC

Modify pc register

GoSearch22's Anti-Analysis Logic

system integrity protection (sip) status detection

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

Two arguments

...but what's the branch target?

call to objc_msgSend

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

(lldb) reg read $x8
x8 = 0x0000000193a5f160  libobjc.A.dylib`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.

Arg 0: self
object method is invoked upon

Arg 1: op
selector of method

```
% lldb 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

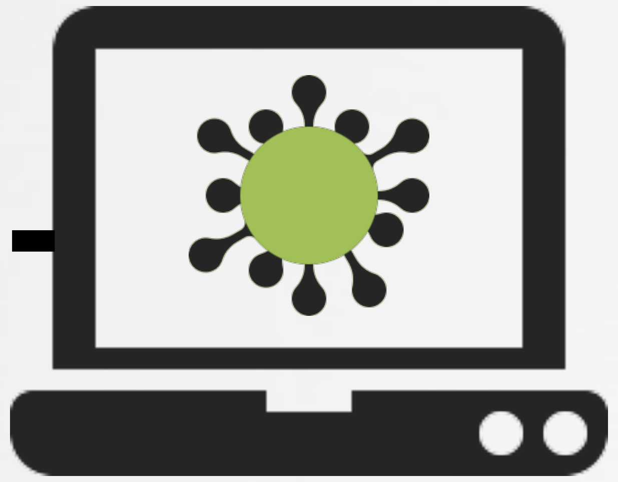
Sets the command arguments that should be used to launch the executable.

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.
```



analysis machine

SIP: disabled?
(malware exits!)

GoSearch22's Anti-Analysis Logic

virtual machine detection

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

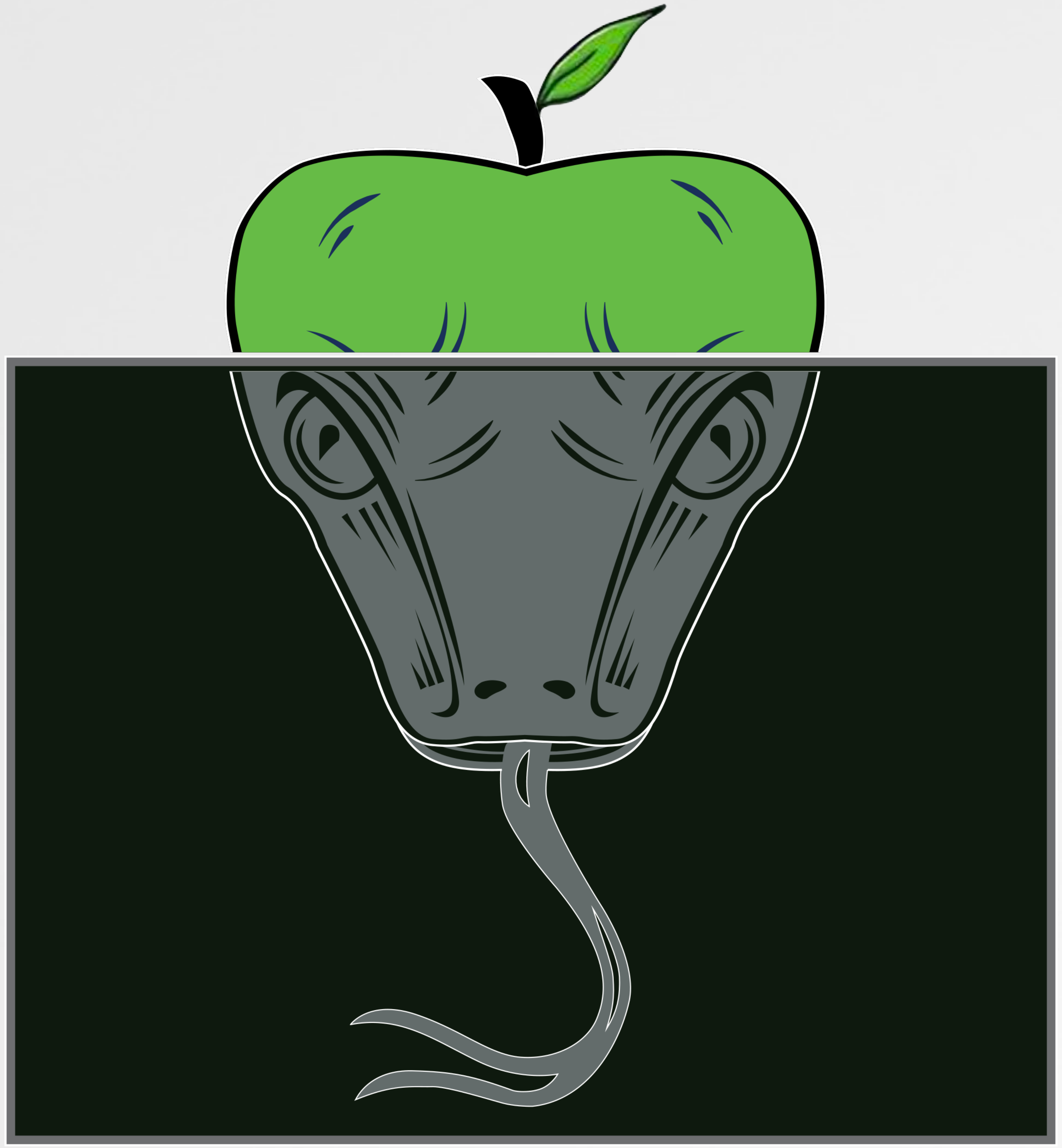
(another) call to obj_msgSend

```
(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

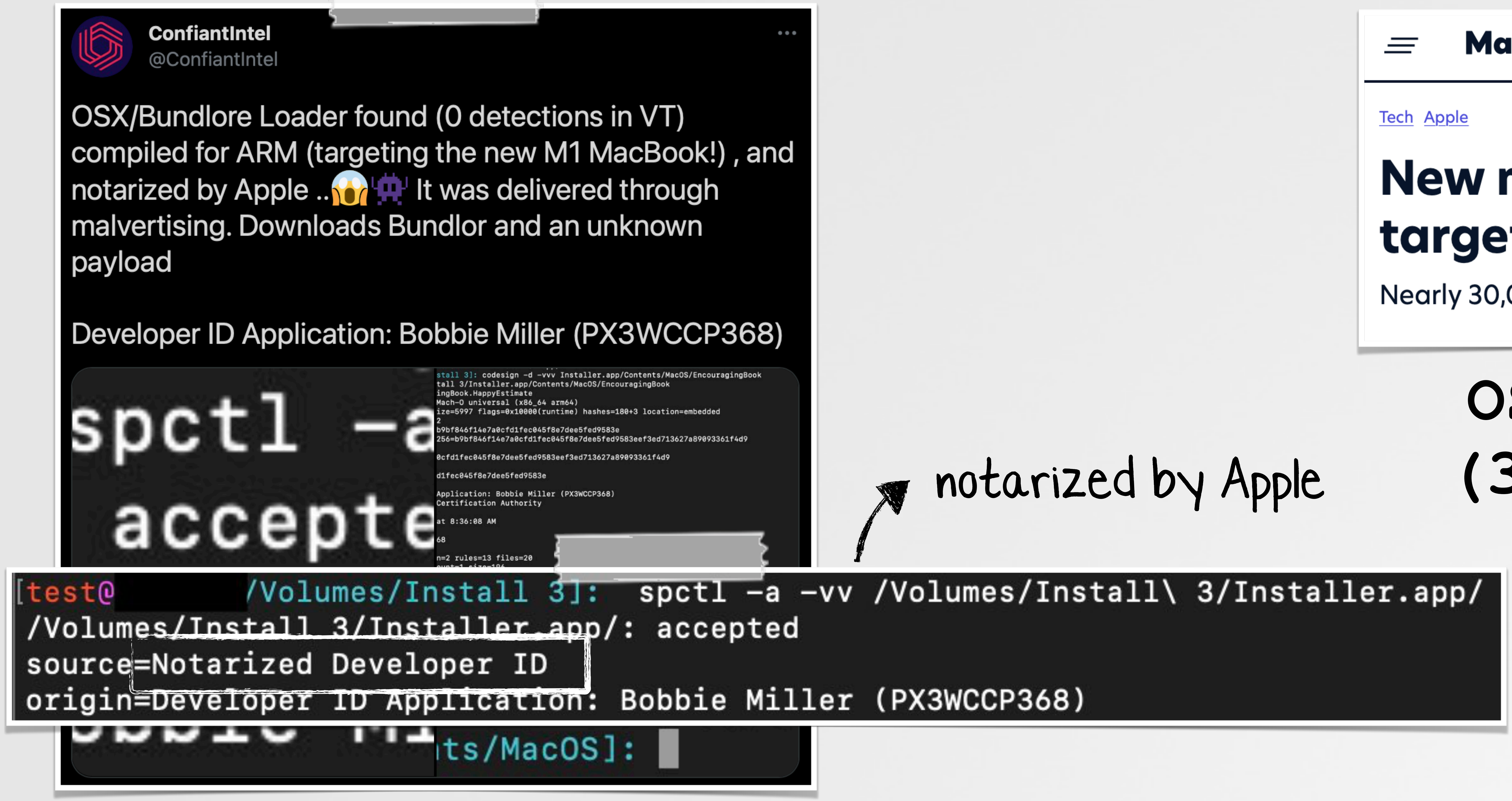
virtual machine detection

Conclusions



...AND MORE!

notarization, infection numbers, etc...



notarized by Apple

OSX.Hydromac
(notarized!)

OSX.SilverSparrow
(30k+ infections!)

"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

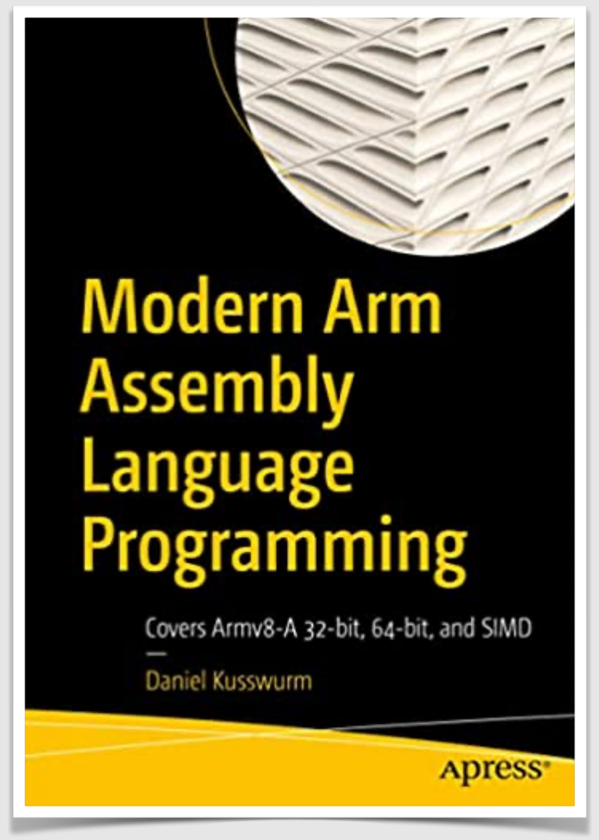
Understanding arm64

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"

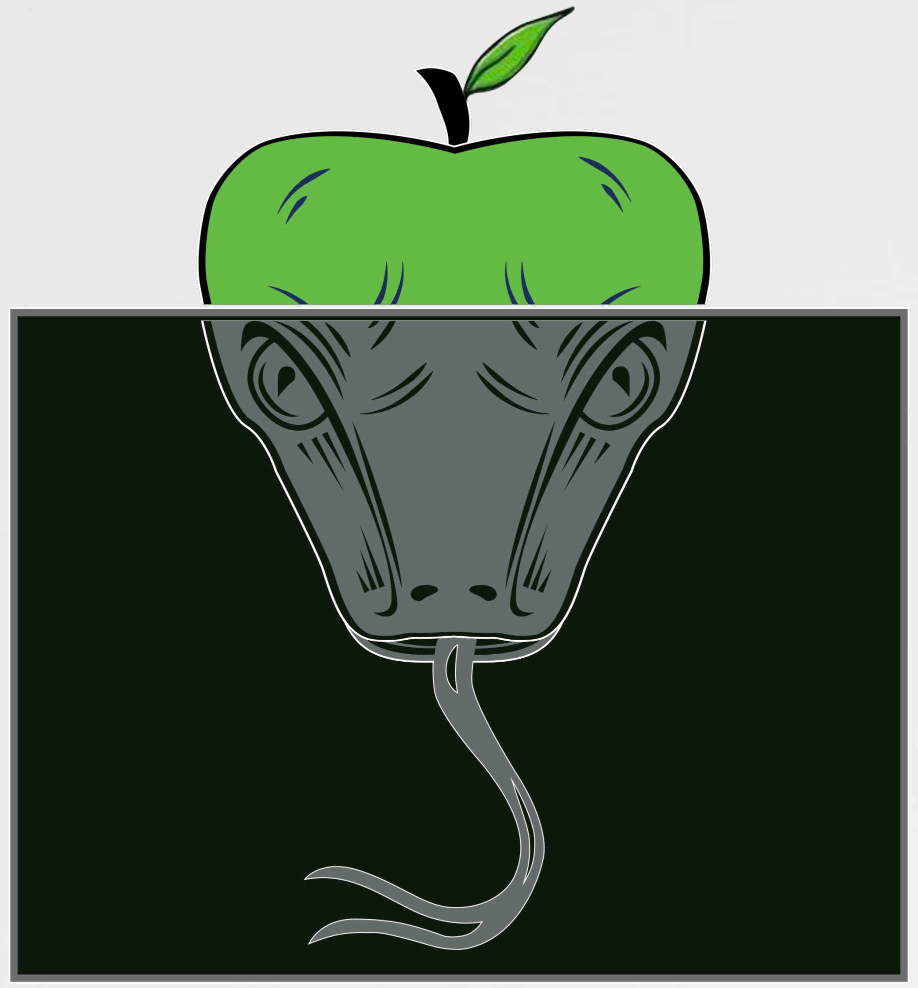
1Password kandji

MOSYLE jamf

SmugMug Guardian Mobile Firewall SecureMac iVerify 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/