# Pishi

Coverage guided macOS KEXT fuzzing.

Meysam Firouzi @R00tkitSMM
POC2024
November 7-8, 2024

# Whoami

- Security Researcher @ MBition - Mercedes-Benz Innovation Lab.
- Focusing on low level stuff.
- Used to be a Windows hacker, now mostly Linux and XNU.

https://R00tkitSMM.github.io

# Agenda

- Fuzzing ImageIO and AppleAVD.
- Why I ended up implementing Pishi.
- Kernel Instrumentation options.
- Structure aware fuzzing.

# ImageIO

ImageIO is Apple's Framework that handles image parsing, which exposes 0click attack surface.

Wednesday, December 15, 2021

## A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution

Posted by Ian Beer & Samuel Groß of Google Project Zero

*We want to thank Citizen Lab for sharing a sample of the FORCEDENTRY exploit with us, and Apple's Security Engineering and Architecture (SEAR) group for collaborating with us on the technical analysis. The editorial opinions reflected below are solely Project Zero's and do not necessarily reflect those of the organizations we collaborated with during this research.*

Tuesday, April 28, 2020

## Fuzzing ImageIO

**Posted by Samuel Groß, Project Zero**

This blog post discusses an old type of issue, vulnerabilities in image format parsers, in a new(er) context: on interactionless code paths in popular messenger apps. This research was focused on the Apple ecosystem and the image parsing API provided by it: the ImageIO framework. Multiple vulnerabilities in image parsing code were found, reported to Apple or the respective open source image library maintainers, and subsequently fixed. During this research, a lightweight and low-overhead guided fuzzing approach for closed source binaries was implemented and is released alongside this blogpost.

ISOSCELES Services Company Contact

## The WebP 0day

Home » Blog

## Exploiting the libwebp Vulnerability, Part 1: Playing with Huffman Code

November 3, 2023 · 2345 words · DARKNAVY | Translations: Zh

▶ Table of Contents

## Vulnerability Localization

# ImageIO

Closed Source macOS binary fuzzing.

IIORawCamera_Reader::testHeader
IIO_Reader_AI::testHeader
IIO_Reader_ASTC::testHeader
IIO_Reader_ATX::testHeader
IIO_Reader_AppleJPEG::testHeader
IIO_Reader_BC::testHeader
IIO_Reader_BMP::testHeader
IIO_Reader_CUR::testHeader
IIO_Reader_GIF::testHeader
IIO_Reader_HEIF::testHeader
IIO_Reader_ICNS::testHeader
IIO_Reader_ICO::testHeader
IIO_Reader_JP2::testHeader
IIO_Reader_KTX::testHeader
IIO_Reader_LibJPEG::testHeader
IIO_Reader_MPO::testHeader
IIO_Reader_OpenEXR::testHeader
IIO_Reader_PBM::testHeader
IIO_Reader_PDF::testHeader
IIO_Reader_PICT::testHeader (macOS only)
IIO_Reader_PNG::testHeader
IIO_Reader_PSD::testHeader
IIO_Reader_PVR::testHeader
IIO_Reader_RAD::testHeader
IIO_Reader_SGI::testHeader (macOS only)
IIO_Reader_TGA::testHeader
IIO_Reader_TIFF::testHeader

Tuesday, April 28, 2020

## Fuzzing ImageIO

**Posted by Samuel Groß, Project Zero**

This blog post discusses an old type of issue, vulnerabilities in image format parsers, in a new(er) context: on interactionless code paths in popular messenger apps. This research was focused on the Apple ecosystem and the image parsing API provided by it: the ImageIO framework. Multiple vulnerabilities in image parsing code were found, reported to Apple or the respective open source image library maintainers, and subsequently fixed. During this research, a lightweight and low-overhead guided fuzzing approach for closed source binaries was implemented and is released alongside this blogpost.

In the end I decided to implement something myself on top of Honggfuzz. The idea for the fuzzing approach is loosely based on the paper: Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing

ie fuzzer then started from a small corpus of around 700 seed images covering the supported image
rmats and ran for multiple weeks. In the end, the following vulnerabilities were identified:

# ImageIO

Closed Source macOS binary fuzzing.

Everyone is fuzzing it now.

Can I beat them?

Jackalope is a customizable, distributed,
coverage-guided fuzzer that is able to work with black-box binaries.

Let’s give it a try, New fuzzer means covering more state spaces.

e fuzzer then started from a small corpus of around 700 seed images covering the supported image
rmats and ran for multiple weeks. In the end, the following vulnerabilities were identified:

Commits on Sep 11, 2020

**Extended TinyInst to macOS**
avniculae committed on Sep 11, 2020

End of commit history for this file

first commit
**ifratric** committed on Dec 15, 2020

`main`

# ImageIO

Closed Source macOS binary fuzzing.

Everyone is fuzzing it now.

Can I beat them?

Wait a minute.

Three new test header functions for different file formats, such as **KTX2**, **WebP**, and **ETC**

### Status of this document

KTX 2.0 ratified by the Khronos Board of Promoters Aug 14th, 2020.

# ImageIO

Closed-source macOS binary fuzzing.

Everyone is fuzzing it now.

Can I beat them?

Wait a minute.

Three new testHeader functions for different file formats. such as **KTX2** and **WebP** and **ETC**.

## Status of this document

KTX 2.0 ratified by the Khronos Board of Promoters Aug 14th, 2020.

Samuel Groß fuzzed OpenEXR, now ImageIO is using Apple's closed-source new implementation of EXR in libAppleEXR.dylib.

One new implementation one and some new file formats.

# ImageIO

Closed-source macOS binary fuzzing.

Everyone is fuzzing it now.

Can I beat them?

To make sure that the coverage-guided fuzzing wouldn't diverge towards other image formats supported by ImageIO.

```
bool isKTX2Header(const uint8_t *buffer, size_t size) {
    const uint8_t ktx2Identifier[12] = {0xAB, 0x4B, 0x54, 0x58, 0x20, 0x32,
                                        0x30, 0xBB, 0x0D, 0x0A, 0x1A, 0x0A};

    if (size < 12) {
        return false; // Buffer is too small to be a KTX2 header
    }

    // Compare the first 12 bytes of the buffer with the KTX2 identifier
    return memcmp(buffer, ktx2Identifier, 12) == 0;
}



bool isEXRHeader(const uint8_t* buffer, size_t size) {
    const uint8_t exrMagicNumber[4] = {0x76, 0x2F, 0x31, 0x01};

    if (size < 4) {
        return false;  // Buffer is too small to be an EXR header
    }

    // Compare the first 4 bytes of the buffer with the EXR magic number
    return memcmp(buffer, exrMagicNumber, 4) == 0;
}
```

# ImageIO

Closed Source macOS binary fuzzing.

Everyone is fuzzing it now.

Can I beat them?

Yes

CVE-2023-32384
CVE-2023-23519
CVE-2023-32372
CVE-2023-27929
CVE-2023-27948
CVE-2023-27947
CVE-2023-42899
CVE-2023-42865
CVE-2023-42862

**ImageIO**

Available for: iPhone 8 and later, iPad Pro (all models), iPad Air 3rd gene
generation and later, and iPad mini 5th generation and later

Impact: Processing an image may lead to arbitrary code execution

Description: A buffer overflow was addressed with improved bounds ch

CVE-2023-32384: Meysam Firouzi @R00tkitsmm working with Trend M

**ImageIO**

Available for: iPhone 8, iPhone 8 Plus, iPhone X, iPad 5th generation, iPac
inch 1st generation

Impact: Processing an image may lead to arbitrary code execution

Description: The issue was addressed with improved memory handling.

CVE-2023-42899: Meysam Firouzi @R00tkitSMM and Junsung Lee

**ImageIO**

Available for: iPhone 8 and later, iPad Pro (all models), iPad Air 3rd generation and later, iPad 5th generation and later, and iPad mini 5th generation and later

Impact: Processing an image may result in disclosure of process memory

Description: An out-of-bounds read was addressed with improved input validation.

CVE-2023-42862: Meysam Firouzi (@R00tkitSMM) of Mbition Mercedes-Benz Innovation Lab

# ImageIO

KTX, WebP, and EXR are file formats.

But HEIF is different

HEIC: HEVC(H.265) in HEIF

AVCI: AVC in HEIF

## HEIF File Extension

| Payload | Extension |
|---|---|
| HEVC | .heic |
| H.264 | .avci |
| any codec | .heif |

Reference: Apple, 503_introducing_heif_and_hevc.pdf and 513_direct_access_to_media_encoding_and_decoding.pdf

Who is decoding H.264, H.265,…?

# AppleAVD

How can I fuzz H.264 and H.265 with Jackalope?

Let’s see what is happening on AppleAVD with DTrace

**DTrace** is a comprehensive dynamic tracing framework

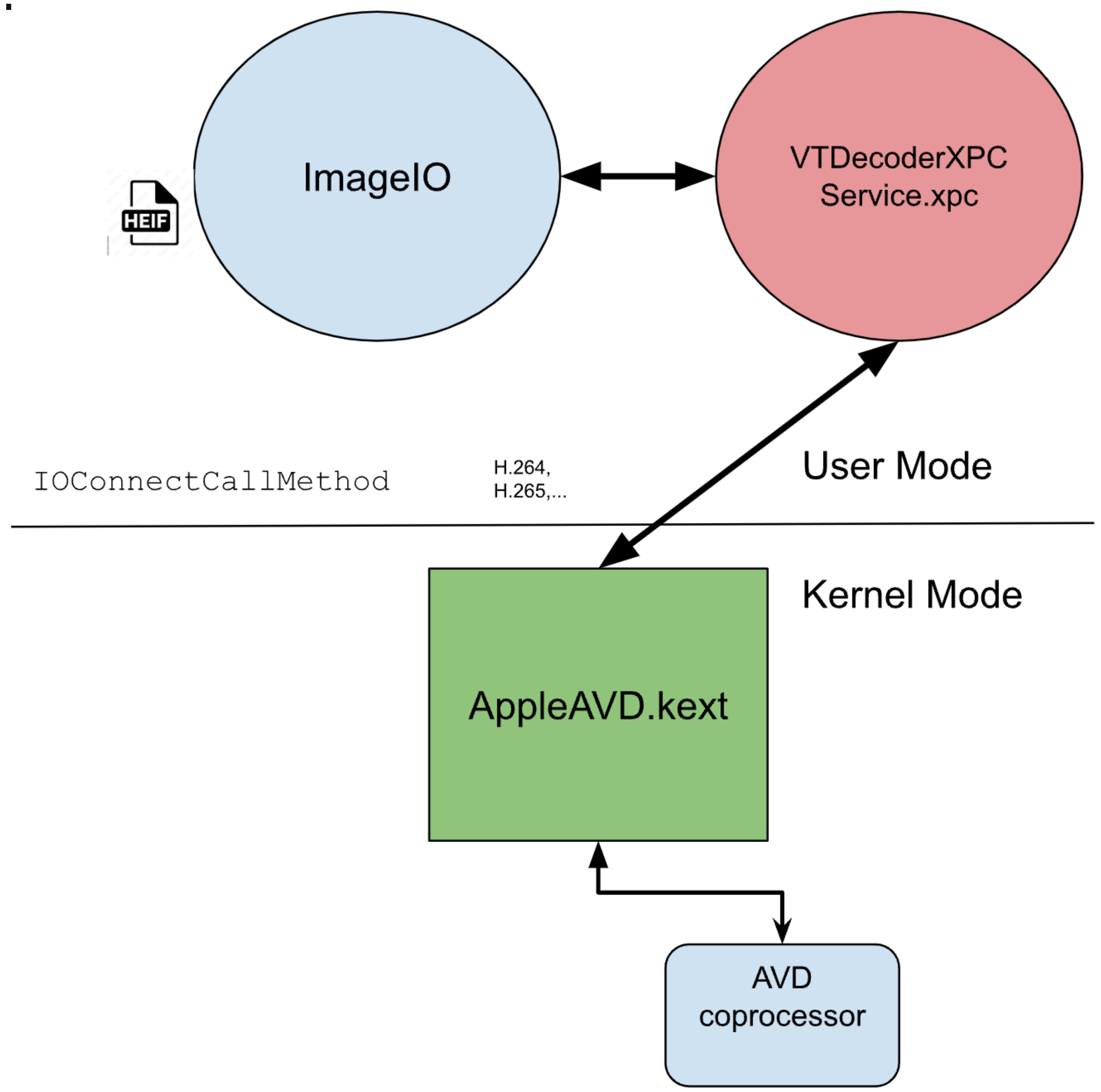
Opening image **HEIC or AVCI will lead to**

```
_ZN8AppleAVD13newUserClientEP4taskPvjPP12IOUserClient:entry execname VTDecoderXPCSe
```

```
c++filt -n  _ZN8AppleAVD13newUserClientEP4taskPvjPP12IOUserClient
AppleAVD::newUserClient(task*, void*, unsigned int, IOUserClient**)

cd /System/Library/Frameworks/VideoToolbox.framework/XPCServices/VT
VTDecoderXPCService.xpc/       VTEncoderXPCService.xpc/
```

# AppleAVD

ImageIO does not talk with AppleAVD directly.



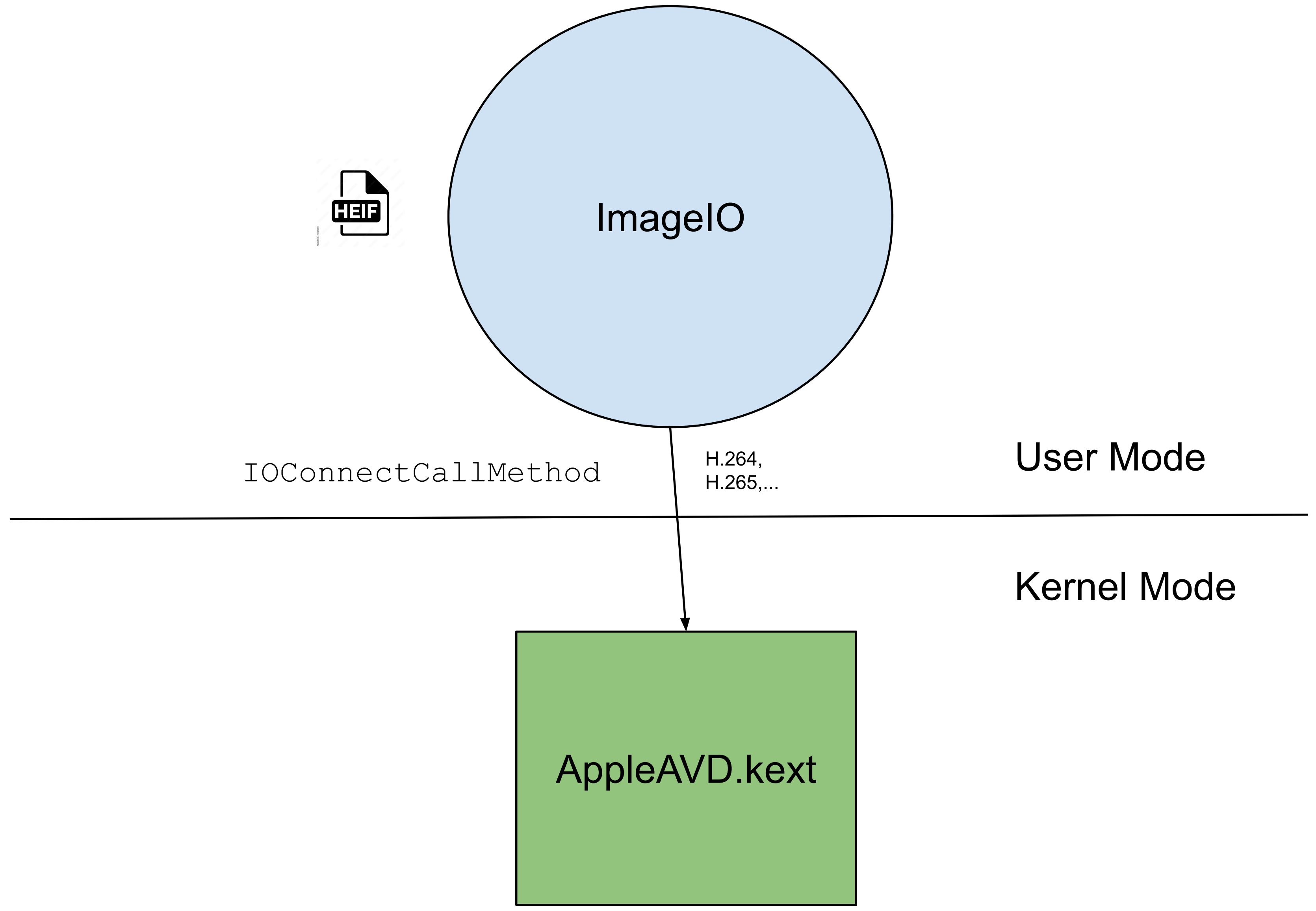
# AppleAVD

Can ImageIO talk with AppleAVD directly?

Ivan: Yes

But we are fuzzing file format if we mutate files.

And payload is deep inside HEIF



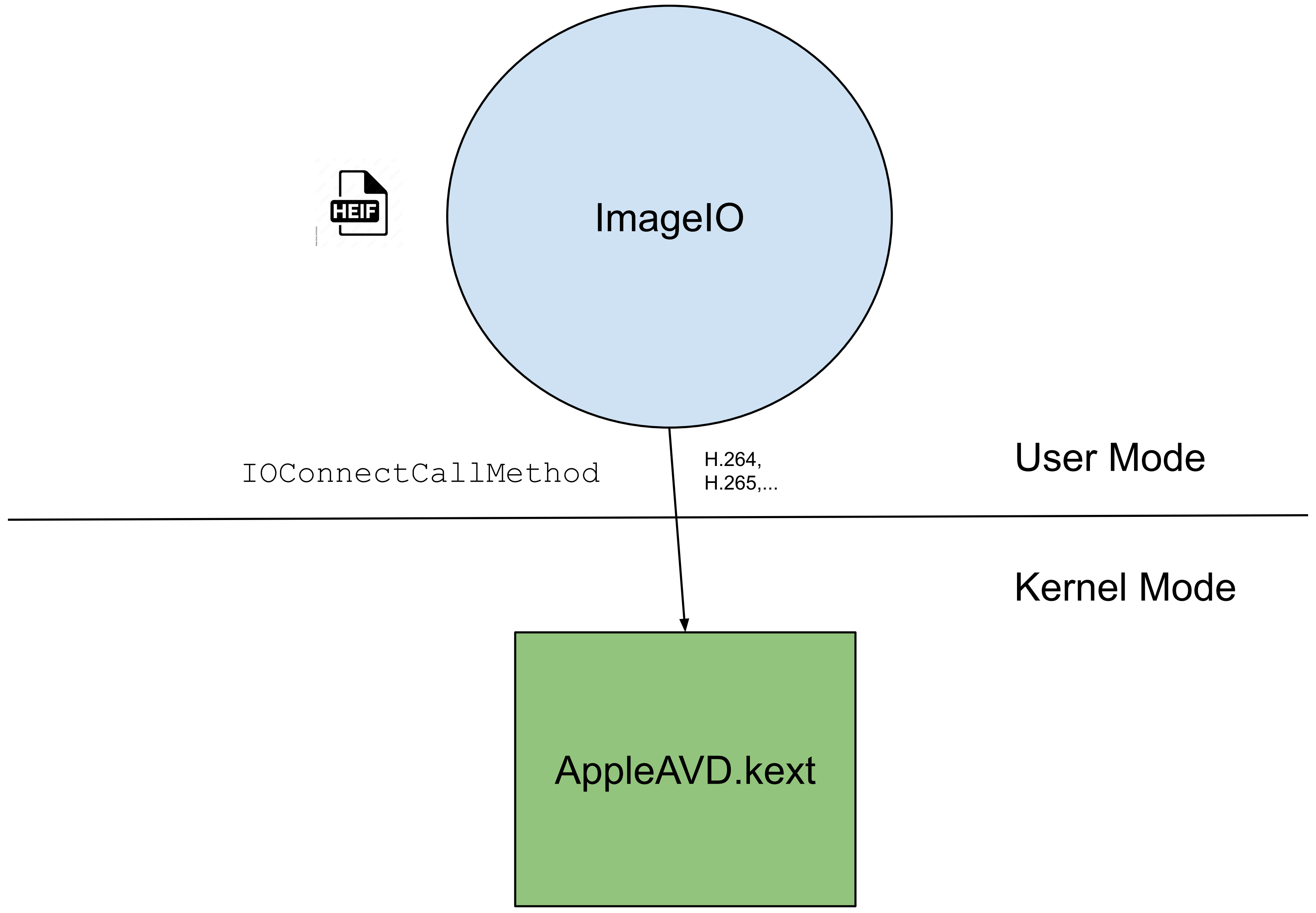


One issue encountered during early testing is that, by default, VideoToolbox creates a separate decoding process, where the decoding actually happens. Thus, a fuzzing harness that just calls video decoding functions won't work well because all the interesting processing will not happen in the harness process. Fortunately, in the VideoToolbox module, a flag called `sVTRunVideoDecodersInProcess` exists, which as the name suggests, causes decoding to take place in the same process. While this flag is not exported, it can also be set by calling the exported function `VTApplyRestrictions` with the argument set to 1. This is what the harness does during initialization.

# AppleAVD

ImageIO is Apple's Framework that handles image parsing, which exposes 0click attack surface.

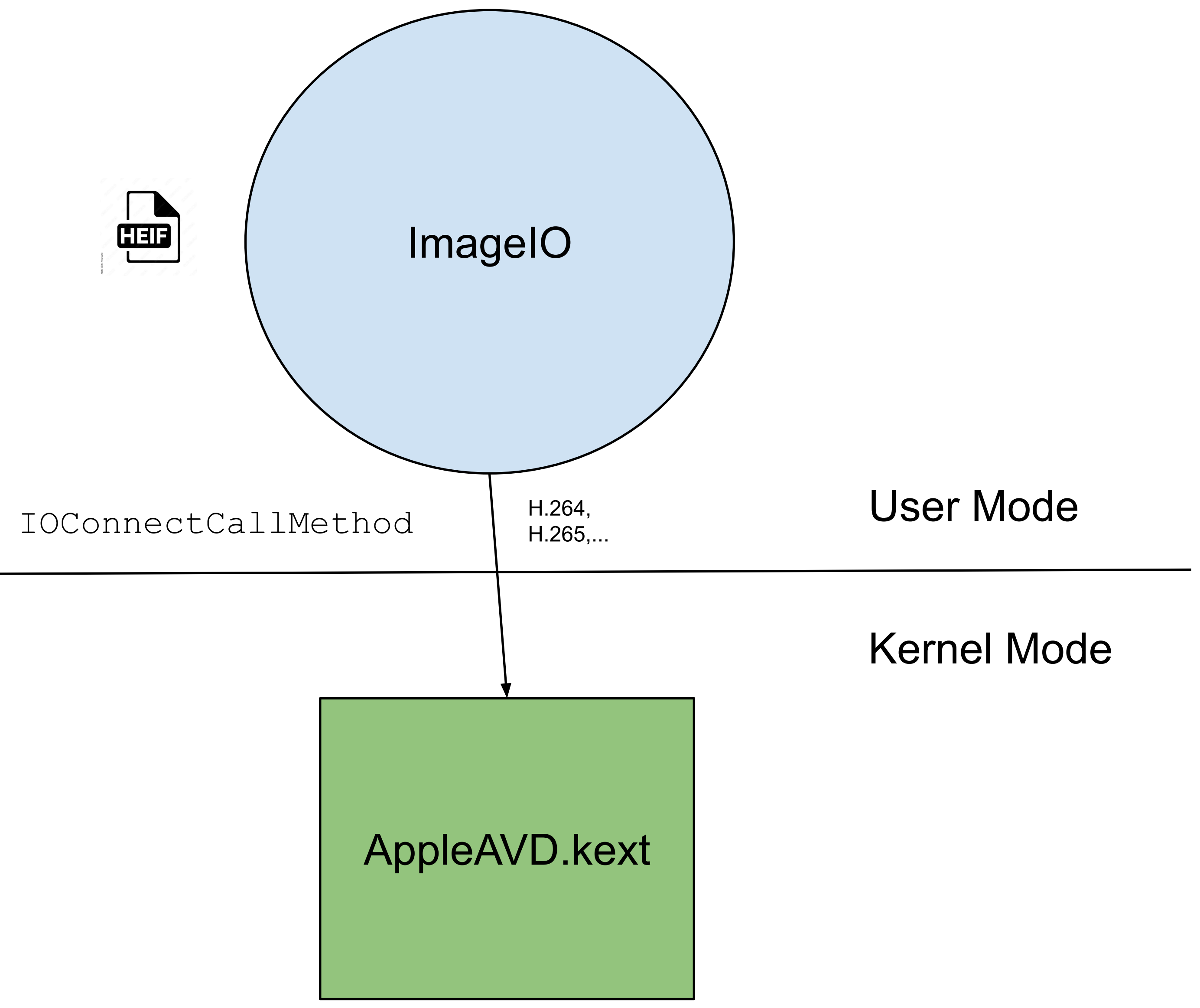
Let's mutate payload just before passing to the kernel.




```
typedef struct interposer {
  void* replacement;
  void* original;
} interpose_t;

__attribute__((used)) static const interpose_t interposers[]
  __attribute__((section("__DATA, __interpose"))) =
    {
      { .replacement = (void*)fake_IOConnectCallMethod,
        .original    = (void*)IOConnectCallMethod
      }
    };
```

```
void flip_bit(void* buf, size_t len){
  if (!len)
    return;
  size_t offset = rand() % len;
  ((uint8_t*)buf)[offset] ^= (0x01 << (rand() % 8));
}

kern_return_t fake_IOConnectCallMethod( ....)
{

  flip_bit(inputStruct, inputStructCnt);

    return IOConnectCallMethod(
      connection,
      selector,
      input,
      inputCnt,
      inputStruct,
      inputStructCnt,
      output,
      outputCnt,
      outputStruct,
      outputStructCntP);
}
```

# AppleAVD




```
panic(cpu 4 caller 0xfffffe0026851cdc): Unaligned kernel data abort. at pc 0xfffffe0026aed514, lr 0xfffffe0026aed5d8 (saved state: 0xfffffe3a396e3200)
      x0:  0x000000000000000e x1:  0xfffffe1002bdc01b  x2:  0x0000000000000000  x3:  0xfffffe3a396e3444
      x4:  0xfffffe3a396e344c x5:  0x000000000002d1f4  x6:  0x0000000000000000  x7:  0xffffffffffffffff
      x8:  0x0000000000000004 x9:  0xfffffe1002bdc01f  x10: 0x0000000000000000  x11: 0x0000000000000002
      x12: 0x0000000000000004 x13: 0x0000000000000000  x14: 0x0000000000000000  x15: 0x0000000000000000
      x16: 0xfffffe0026aeda90 x17: 0xfffffe0026aed9fc  x18: 0x0000000000000000  x19: 0xfffffe1b40e90000
      x20: 0x0000000000000000 x21: 0x0000000000000000  x22: 0xfffffe1002bdc000  x23: 0x0000000000000001
      x24: 0x0000000000000000 x25: 0xfffffe1002bdc024  x26: 0x000000000000001b  x27: 0x00000000000008b0
      x28: 0x000000000000001b fp:  0xfffffe3a396e3610  lr:  0xfffffe0026aed5d8  sp:  0xfffffe3a396e3550
      pc:  0xfffffe0026aed514 cpsr: 0x60401208         esr: 0x96000021          far: 0xfffffe1002bdc01b

Debugger message: panic
Memory ID: 0x6
OS release type: User
OS version: 23C71
Kernel version: Darwin Kernel Version 23.2.0: Wed Nov 15 21:53:34 PST 2023; root:xnu-10002.61.3~2/RELEASE_ARM64_T8103
Fileset Kernelcache UUID: 6DAC2CF8E68E8F436296A697E29AAD44
Kernel UUID: E245D804-1FA3-31E2-90BC-B4DF75B2129E
Boot session UUID: 52885412-0864-4DFF-8E9E-36C3C7BC8B88
iBoot version: iBoot-10151.61.4
```

### AppleAVD

Available for: iPhone XS and later, iPad Pro 13-inch, iPad Pro 12.9-inch 2nd generation and later, iPad Pro 10.5-inch, iPad Pro 11-inch 1st generation and later, iPad Air 3rd generation and later, iPad 6th generation and later, and iPad mini 5th generation and later

Impact: An app may be able to cause unexpected system termination

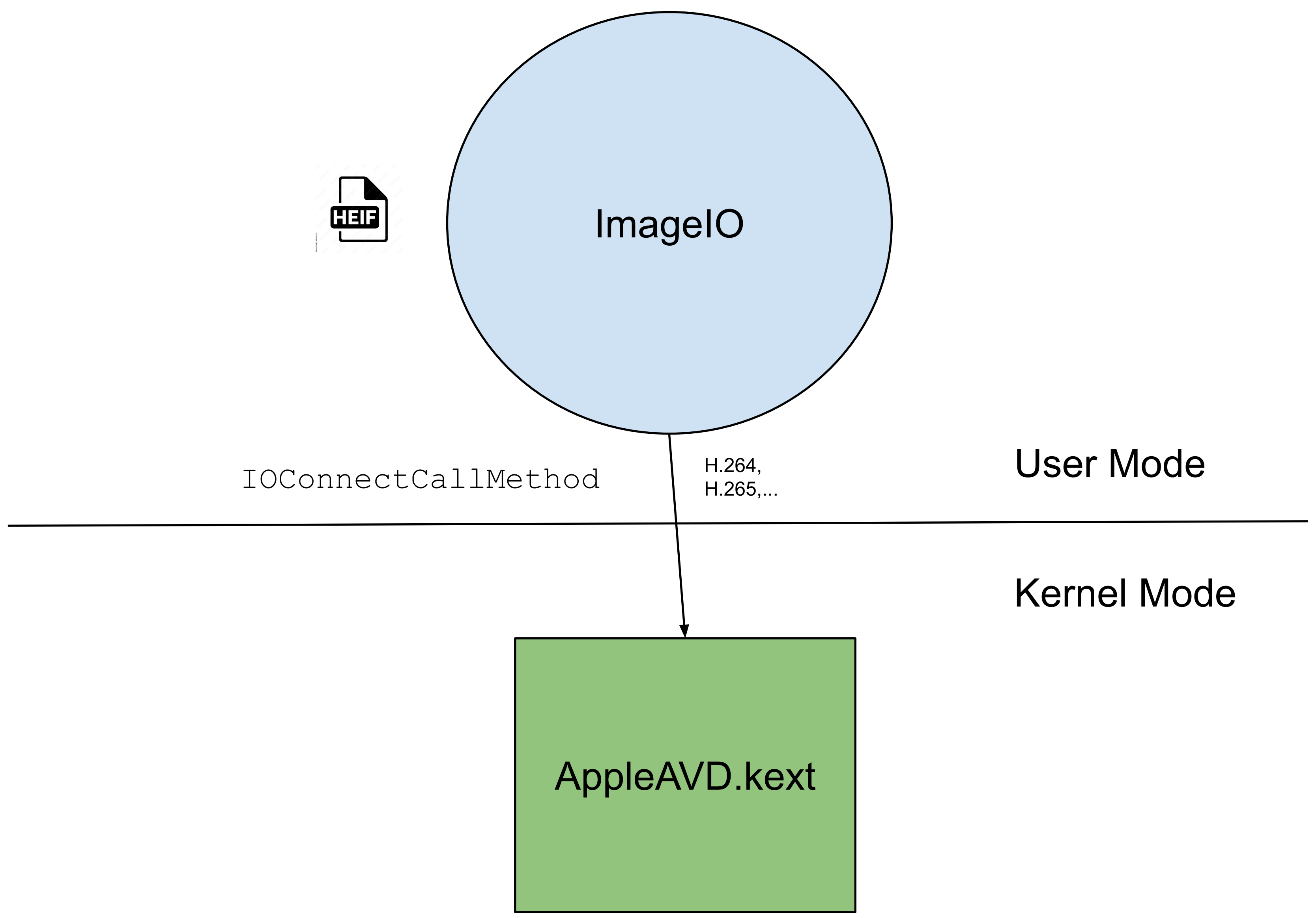
Description: The issue was addressed with improved memory handling.

CVE-2024-27804: Meysam Firouzi (@R00tkitSMM)

Entry updated May 15, 2024

# AppleAVD

What we are mutating?
let's talk with AppleAVD directly.
But we have no clue what functions or BBs have been covered.

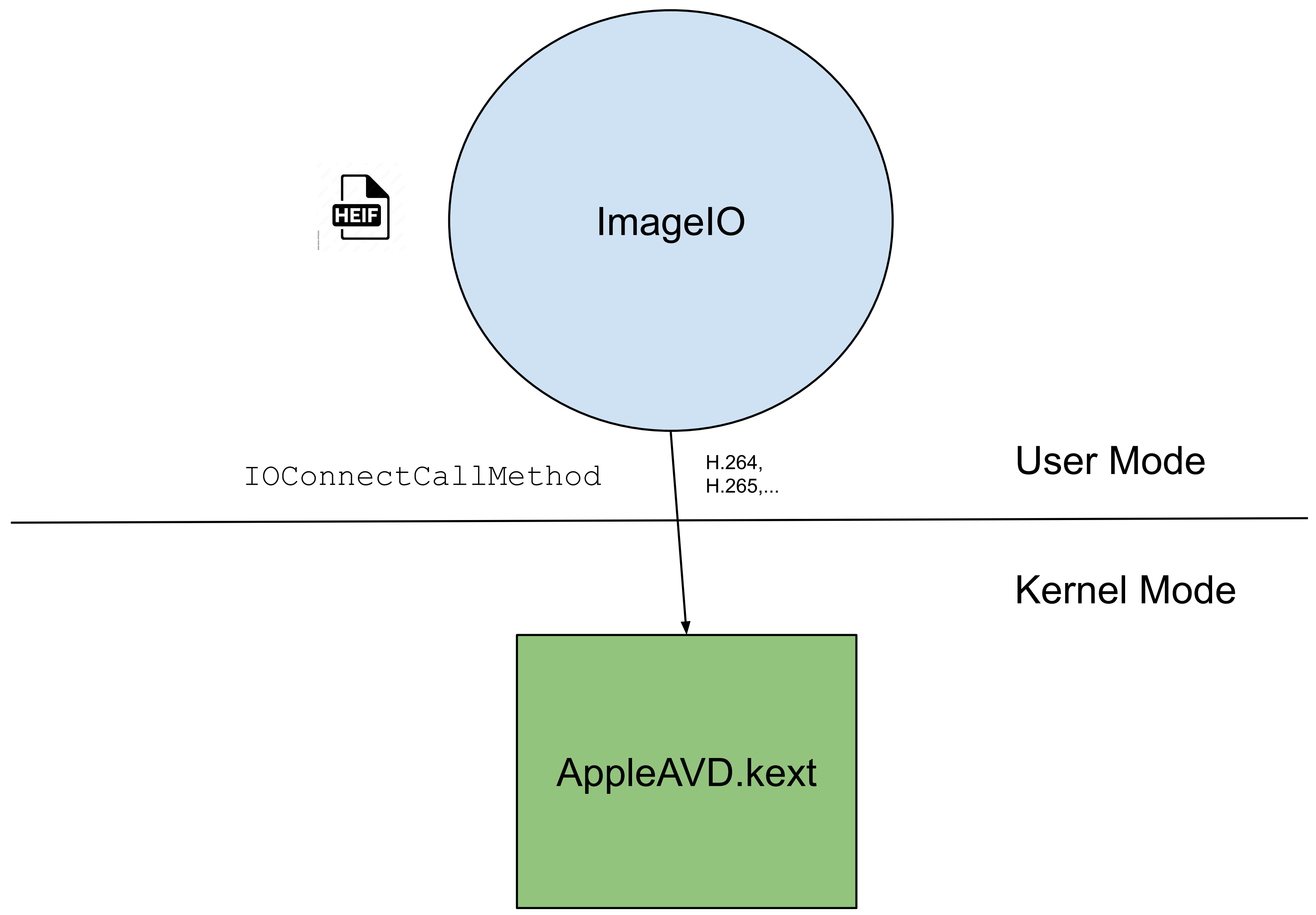



```
CFMutableDictionaryRef matching = IOServiceMatching("AppleAVD");
err = IOServiceGetMatchingServices(kIOMasterPortDefault, matching, &iterator);
io_service_t service = IOIteratorNext(iterator);
err = IOServiceOpen(service, mach_task_self(), stype, &conn);
IOConnectCallMethod( // createDecoder
    conn,
    0,
    inputScalar,
        inputScalarCnt,
        inp,
        0xd8,
        outputScalar,
        &outputScalarCnt,
        outputStruct,
        &out_num);
```

**This is part of a POC by Natalie Silvanovich.**

# AppleAVD

This is good.
But we have no clue what are have covered.




```
CFMutableDictionaryRef matching = IOServiceMatching("AppleAVD");
err = IOServiceGetMatchingServices(kIOMasterPortDefault, matching, &iterator);
io_service_t service = IOIteratorNext(iterator);
err = IOServiceOpen(service, mach_task_self(), stype, &conn);
IOConnectCallMethod( // createDecoder
    conn,
    0,
    inputScalar,
        inputScalarCnt,
        inp,
        0xd8,
        outputScalar,
        &outputScalarCnt,
        outputStruct,
        &out_num);
```

**Part of a POC by Natalie Silvanovich.**

DTrace again.

```
[meysam@meysams-MacBook-Air ~ %
[meysam@meysams-MacBook-Air ~ %
[meysam@meysams-MacBook-Air ~ % sudo dtrace -l | grep AppleAVD | wc -l
    1501
[meysam@meysams-MacBook-Air ~ %
[meysam@meysams-MacBook-Air ~ %
```

Just fifteen hundred.
anyway this is not an input for fuzzer.

# AppleAVD

## Fuzzing AppleAVD

Willy R. Vasquez
https://wrv.github.io › ... PDF

Finding and Exploiting Vulnerabilities in H.264 Decoders

by WR Vasquez · Cited by 1 — Our **fuzzing** setup consisted of (1) generating a batch of. 100 videos on a host machine, (2) transferring them to the iOS device under test ( ...
18 pages

Project Zero — AppleAVD

| P | TYPE | TITLE |
|---|---|---|
| P2 | Bug | AppleAVD: Memory Corruption in AppleAVDUserClient::decodeFrameFig |
| P2 | Bug | AppleAVD: Missing surface lock in deallocateKernelMemoryInternal |
| P2 | Bug | AppleAVD: Overflow in AVC_RBSP::parseSliceHeader ref_pic_list_modification |

### Cinema time!

#### Abstract

Media parsing is known as one of the weakest components of every consumer system. It often o
security requirements, such as attack surface minimization, compartmentalization, and privilege
interesting case for two different reasons. First, instead of running in usermode, a considerable p
kernel to additional remote attack vectors. Second, recent anonymous reports suggest that Apple
depth, covering video decoding subsystem internals, analysis of vulnerabilities, and ways to exploit t

#### Resources

Slides: hexacon2022_AppleAVD.pdf

# AppleAVD

Fuzzing AppleAVD

AppleAVD is closed source.

Dumb fuzzing won’t give us anything. we need a **feedback-driven fuzzing.**

What are the macOS kernel instrumentation options for M1/Apple Silicon?

# KEXT/XNU Fuzzing

macOS kernel instrumentation options for M1/Apple Silicon:

macOS is a mix of open source and closed source components.

Open Source part:

XNU: KSANCOV, KASAN kernel binary in KDK does not have **KSANCOV**. And I don't like building XNU with KCOV.

XNU: SockFuzzer, XNU kernel is compiled as a library and run within a custom user space environment.

BUT AppleAVD is Closed source.

# KEXT/XNU Fuzzing

macOS kernel instrumentation options for M1/Apple Silicon:

How to instrument closed source KEXTs?

**Hardware-based instrumentation:**

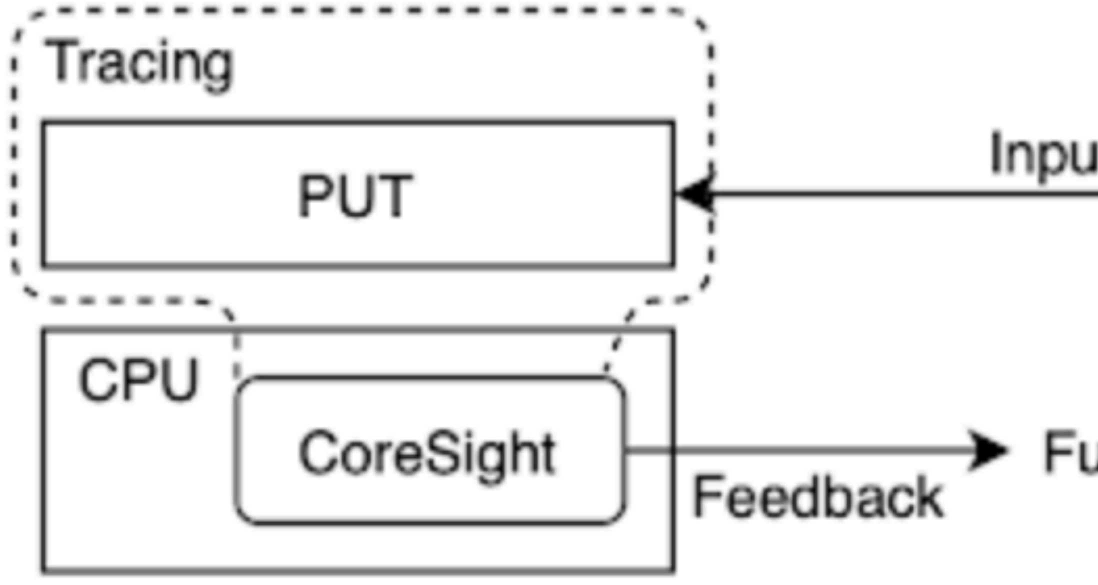
Intel CPUs:
Intel-PT is a technology available in modern Intel CPUs that allows efficient tracing of all the instructions executed by a process.
kAFL relies on a special CPU feature, i.e., Intel Processor Trace (Intel-PT), to collect the code coverage information

But M1/Apple Silicon is Arm based.

# KEXT/XNU Fuzzing

macOS kernel instrumentation options for M1/Apple Silicon:

## Hardware based instrumentation:




(b-2) CoreSight mode

ARMored CoreSight: Towards Efficient Binary-only Fuzzing

ARM CPUs:
Coresight is an umbrella of technologies allowing for the debugging of ARM based SoC.
It includes solutions for JTAG and HW-assisted tracing.

**CoreSight** is a set of hardware features designed to enable system debugging, profiling, and tracing.
important components of CoreSight are the **ETM** (Embedded Trace Macrocell) and **ETR** (Embedded Trace Router)

# KEXT/XNU Fuzzing

**CoreSight** is a set of hardware features designed to enable system debugging, profiling, and tracing.
Two important components of CoreSight are the **ETM** (Embedded Trace Macrocell) and **ETR** (Embedded Trace Router)

```
/*
 * CoreSight debug registers
 */
#define CORESIGHT_ED  0
#define CORESIGHT_CTI 1
#define CORESIGHT_PMU 2
#define CORESIGHT_UTT 3 /* Not truly a
```


**But ETM** and **ETR** are not available in Apple Silicon.
or they are just undocumented:

[KTRW: The journey to build a debuggable iPhone]

**Operation Triangulation**

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Loading KEXT into user mode with a custom Mach-O loader
https://github.com/pwn0rz/fairplay_research/tree/master Implements a loader.

partially with extracted IDA decompiler pseudocode

https://github.com/taviso/loadlibrary, to load dll in Linux

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Instrumentations for binary-only fuzzing are categorized into:

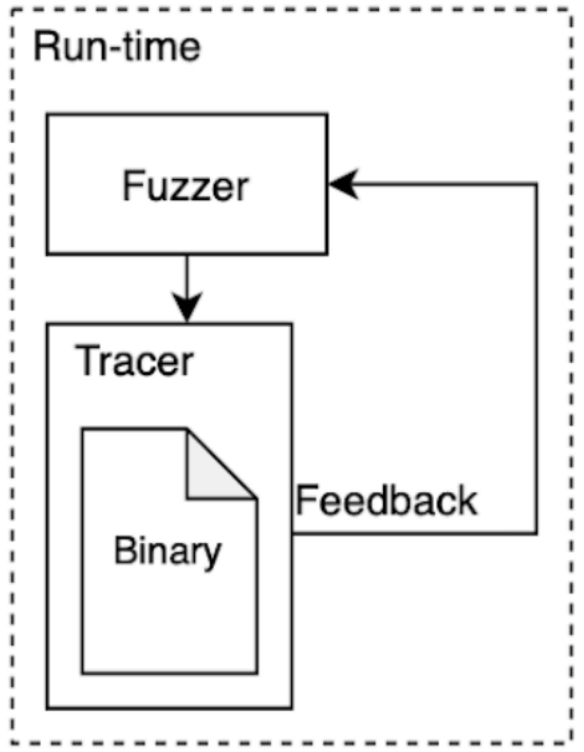
**Dynamic instrumentation:**

Inserts the code for generating feedback into the target program at run time.

With breakpoint or like Jackalope binary rewriting.
feasible but **difficult.**

No breakpoint in Apple Silicon.

Anyway, it needs two devices.




(b) Dynamic Instrumentation

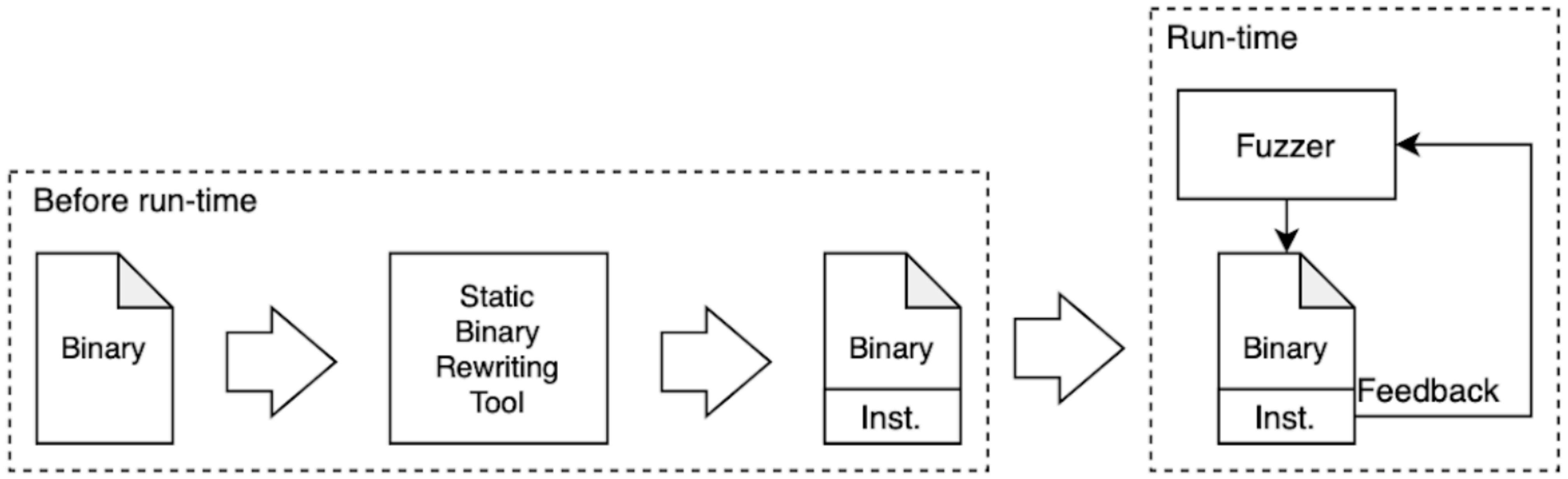
ARMored CoreSight: Towards Efficient Binary-only Fuzzing

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

**Static instrumentation:**
statically rewriting target binaries.

I decided to investigate on this one.




(a) Static Instrumentation

ARMored CoreSight: Towards Efficient Binary-only Fuzzing

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation or Binary rewriting

What do we have to study?

**Retrowrite**: a static binary rewriter for x64 and aarch64
**StochFuzz**: A New Solution for Binary-only Fuzzing
**ArmWrestling**: Efficient binary rewriting for aarch64. which contains IL lifting
**ARMore**: Pushing Love Back Into Binaries
More and more talks

Great talks but they are mostly about user mode binaries. And Linux ELF files
existing methods have fundamental limitations when applied to macOS KEXTs.

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

## Next 2 days:

How to load KEXT?

Do hardware mitigations (KTRR,…) allow me to patch memory in M1?

How to fuzz KEXT?

KextFuzz 🤨

kext binary rewrite

All Videos Images News Web Books Finance

Tutorial Example Mac Github

IEEE Computer Society
https://www.computer.org › csdl › journal › 2024/04

KextFuzz: A Practical Fuzzer for macOS Kernel ...

by T Yin · 2024 — KextFuzz patches the target **kext** via static **binary rewriting** before fuzzing it. Compared with the original **kext**, the patched **kext** (the **kext**” in Fig. 6) is ...

GitHub
https://github.com › vul337 › KextFuzz

Code of KextFuzz: Fuzzing macOS Kernel EXTensions on ...

The ./**rewrite** directory contains the code to do **kext** instrumentation and entitlement patch. Step 1. Get a patched **kext**. Note: edit ./**rewrite**/config.json to ...

USENIX
https://www.usenix.org › usenixsecurity23-yin PDF

KextFuzz: Fuzzing macOS Kernel EXTensions on Apple ...

by T Yin · 2023 · Cited by 2 — With the novel static **binary rewriting** method,. **KextFuzz** can track code coverage and find 6X more crashes than a black-box baseline fuzzing ...
17 pages

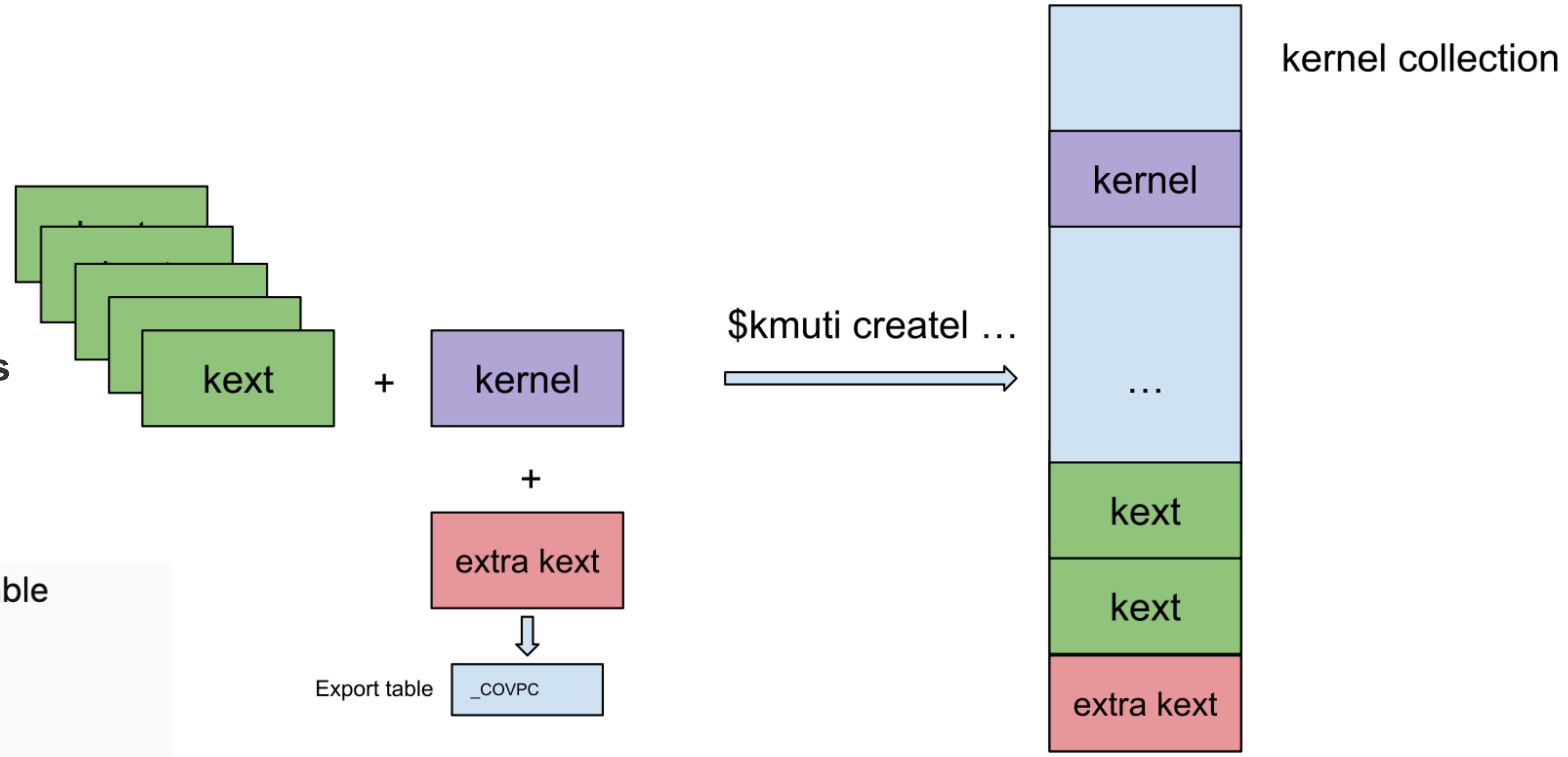
# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

KextFuzz: Fuzzing macOS Kernel EXTensions on Apple Silicon via Exploiting Mitigations

How to load KEXT?




What is Kext Collection?

In macOS 11 or later Apple has changed its previous scheme of prelinked kernelcaches and Loadable kernel module, to three prelinked kernel collections blobs:

- The Boot Kext Collection (BKC), contains the kernel itself, and all the major system kernel extensions required for a Mac to function.
- The System Kext Collection (SKC), This contains all the other system kernel extensions, which are loaded after booting with the BKC.
- The Auxiliary Kext Collection (AKC), is built and managed by the service kernelmanagerd. This contains all installed third-party kernel extensions, and is loaded after the other two collections.

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

What are they instrumenting?

How they are instrumenting?

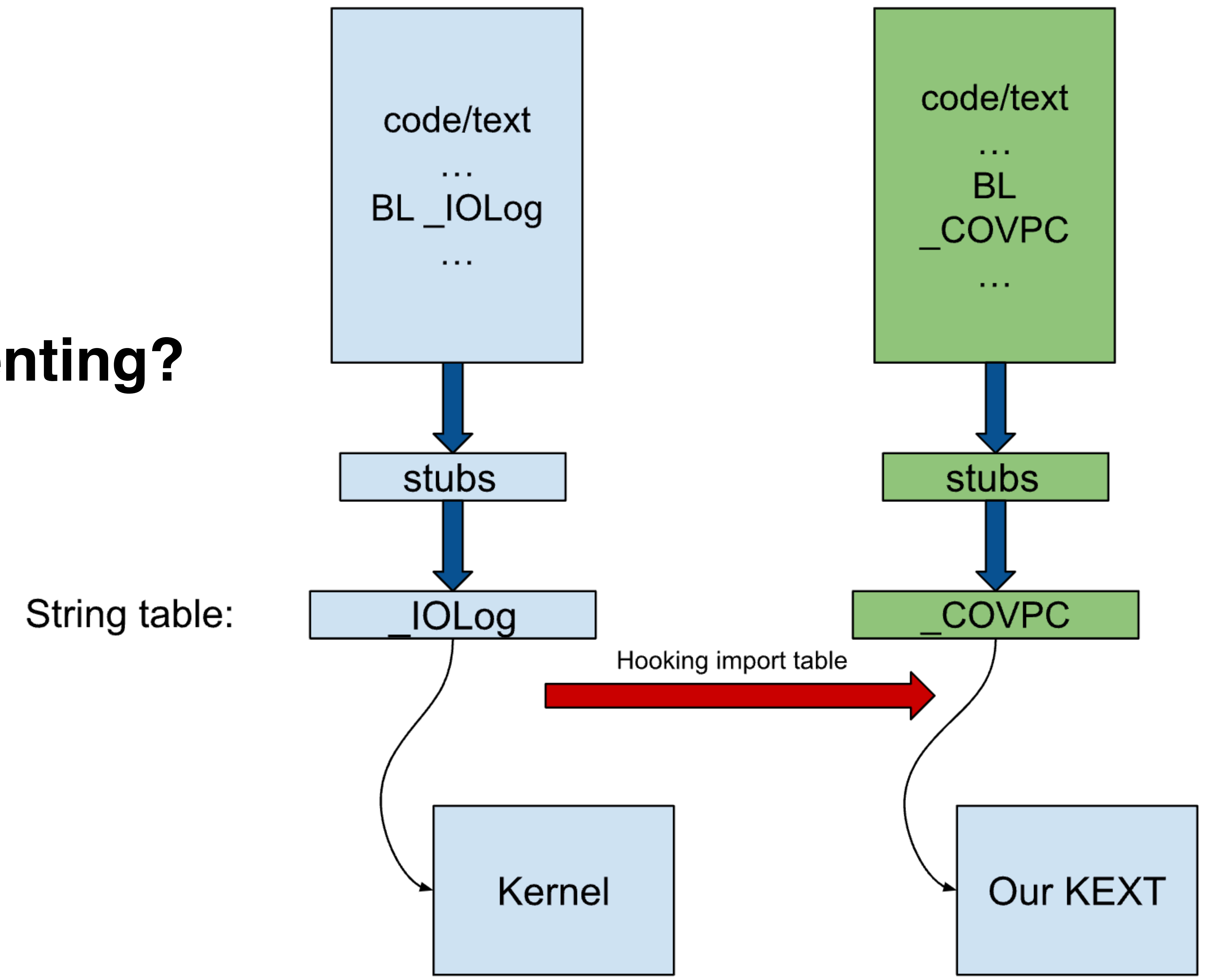
# KEXT/XNU Fuzzing

Static instrumentation:
Binary rewriting

**How they are instrumenting?**

IAT hooking

Replacing a function name in a KEXT, with a string of another function( with exactly same size)



# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

### What are they instrumenting?

Where can they put this BL instructions
without corrupting the original behavior?

1- XPACD instructions

- `XPAC*` instructions remove a pointer's PAC and restore the original value without performing verification.

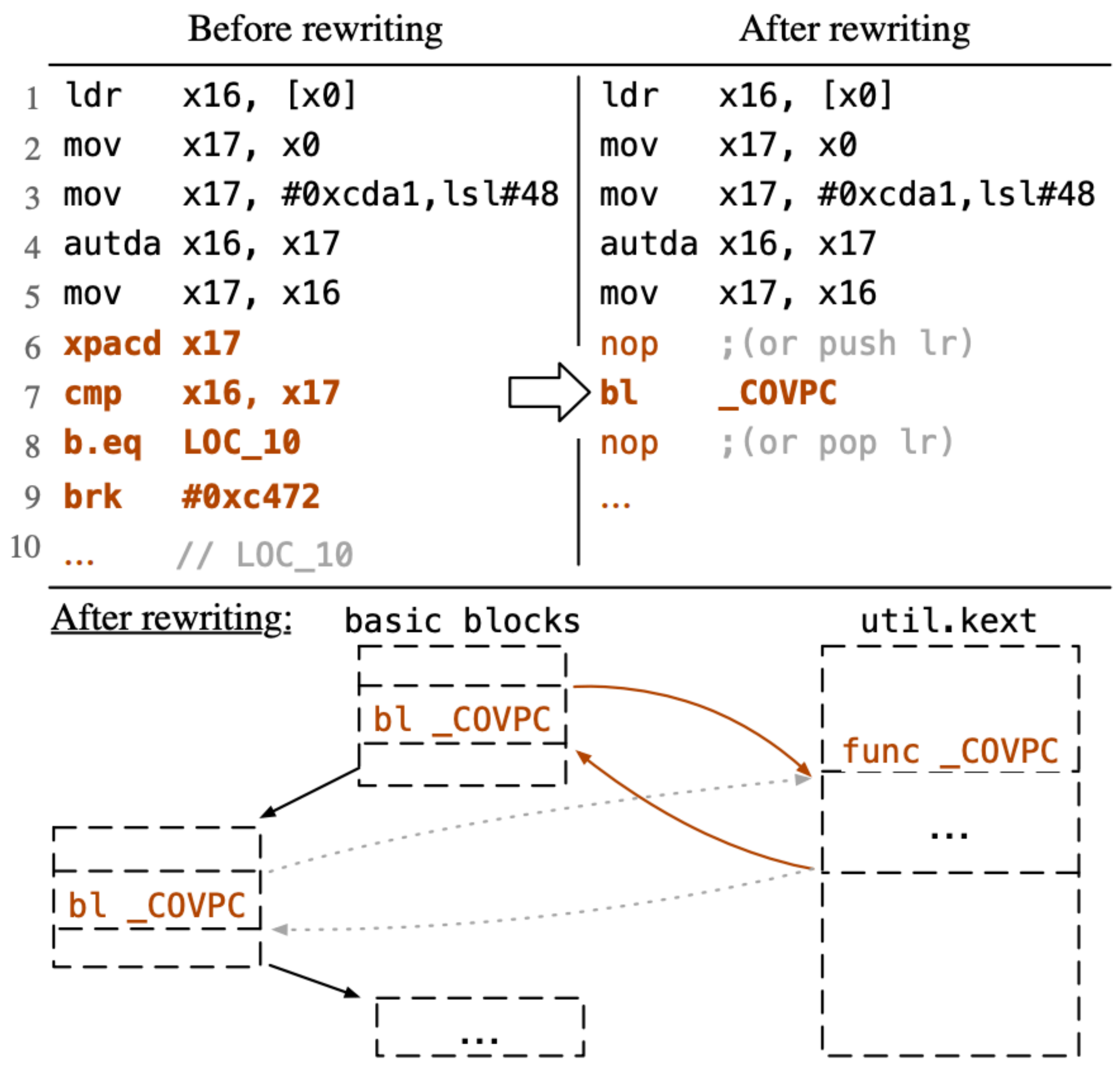



Image by KextFuzz paper.

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

Compiler emitted code for each vtable access.

XPACD instruction can be replaced by a BL.

```
9738: aa0803f1      mov x17, x8 # V_table
973c: f2f97d71      movk    x17, #0xcbeb, lsl #48
9740: dac11a30      autda   x16, x17 # authentica V_table into x16
# If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address.
# If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.
9744: aa1003f1      mov x17, x16 # move x16 into x17
9748: dac147f1      xpacd   x17 # strip key from x17
974c: eb11021f      cmp x16, x17 # compare x16 and x17 to see if they are equal.  equal == means autda was successful
9750: 54000040      b.eq    0x9758 <__ZN20IOSurfaceSharedEvent25signal_completed_internalEyb+0x88>
9754: d4388e40      brk #0xc472
```

It's not part of the program logic.

Calling into a **imported function** is through **__auth_stubs** and it's clobbering X16, X17 and current LR.
we will answer this later.

```
00000000000360d8 <__auth_stubs>:
   360d8: d0000031      adrp    x17, 0x3c000 <_zalloc_flags+0x3c000>
   360dc: 91000231      add x17, x17, #0x0
   360e0: f9400230      ldr x16, [x17]
   360e4: d71f0a11      braa    x16, x17
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

```
## instrument by replacing x30 pa (PACIBSP) instuction, not stable in some cases ##
kext_bytes = instrument_x30_pa(kext_bytes, fileoff, filend, stub_addr)
```

Part of KextFuzz code

Where can they put this BL instructions without corrupting the original behavior?

2- PACIBSP instructions

```
PACIASP
SUB sp, sp, #0x40
STP x29, x30, [sp,#0x30]
ADD x29, sp, #0x30
....
....
LDP x29,x30,[sp,#0x30]
ADD sp,sp,#0x40
AUTIASP
RET
```

prologue

epilogue

Again this is not part of the program logic.

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

This is good but not enough.

KextFuzz can instrument kexts at basic block granularity **roughly** because the kexts are developed in C++ and widely use PA instructions to protect return addresses and indirect calls. In addition, the PA instructions distribute at different points of the program.

Part of KextFuzz paper.

```
if( k_buffer[0] =='M' )
    if( k_buffer[1] =='E' )
        if( k_buffer[2] =='Y' )
            if( k_buffer[3] =='S' )
                if( k_buffer[4] =='A' )
                    if( k_buffer[5] =='M' )
                        if( k_buffer[6] =='6' )
                            if( k_buffer[7] =='7' )
                                if( k_buffer[8] =='8' )
                                    if( k_buffer[9] =='9' ) {
                                        printf("boom!\n");
                                        int* p = (int*)0x41414141;
                                        *p = 0x42424242;
            }
```

**All functions + ALL XPACD instructions, is not roughly, it’s barely.**

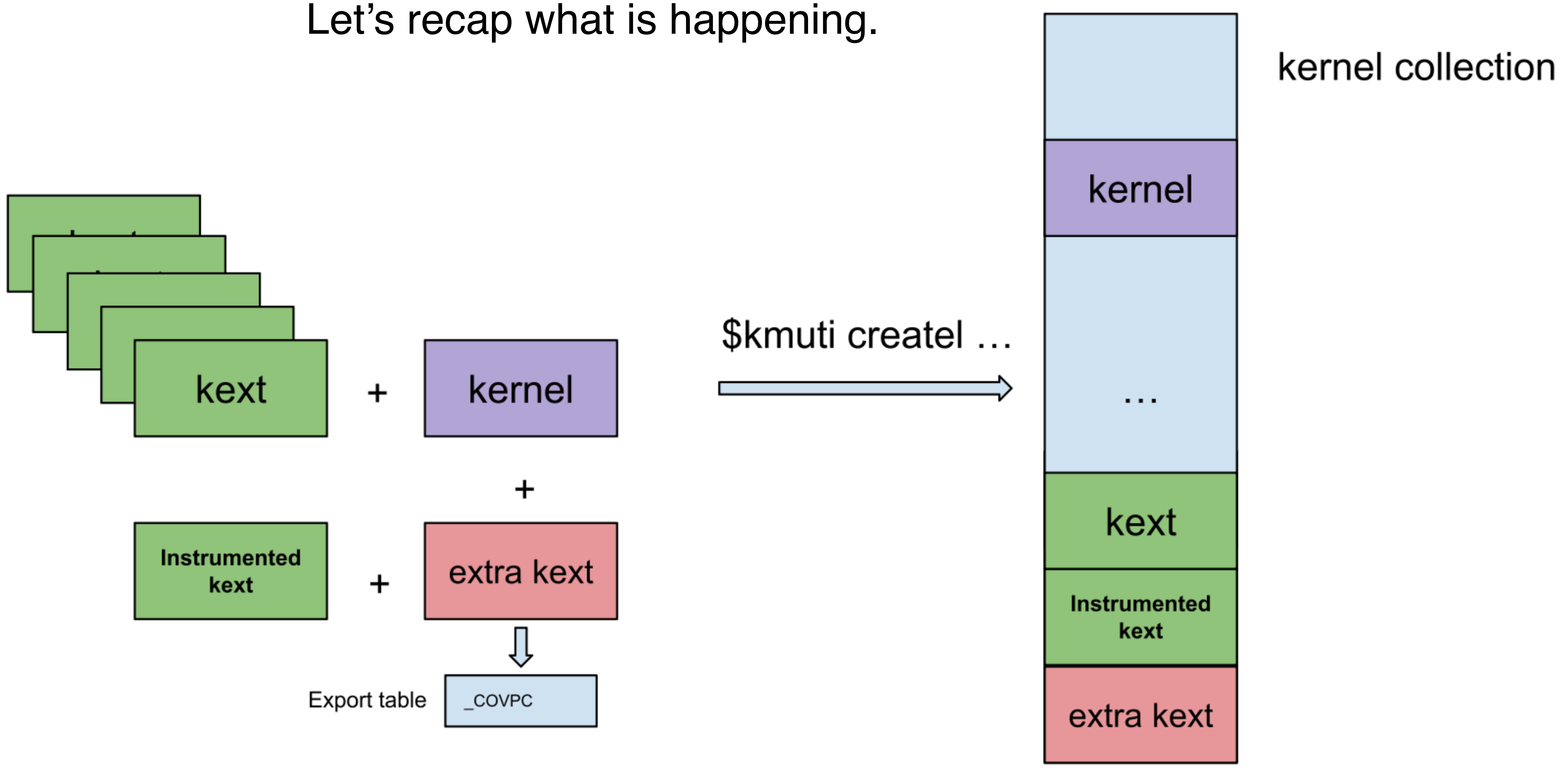
```
meysam@meysams-MacBook-Air Pishi %
meysam@meysams-MacBook-Air Pishi % objdump    --disassemble  IOSurface | grep -i XPACD | wc -l
     189
meysam@meysams-MacBook-Air Pishi %
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

Let’s recap what is happening.

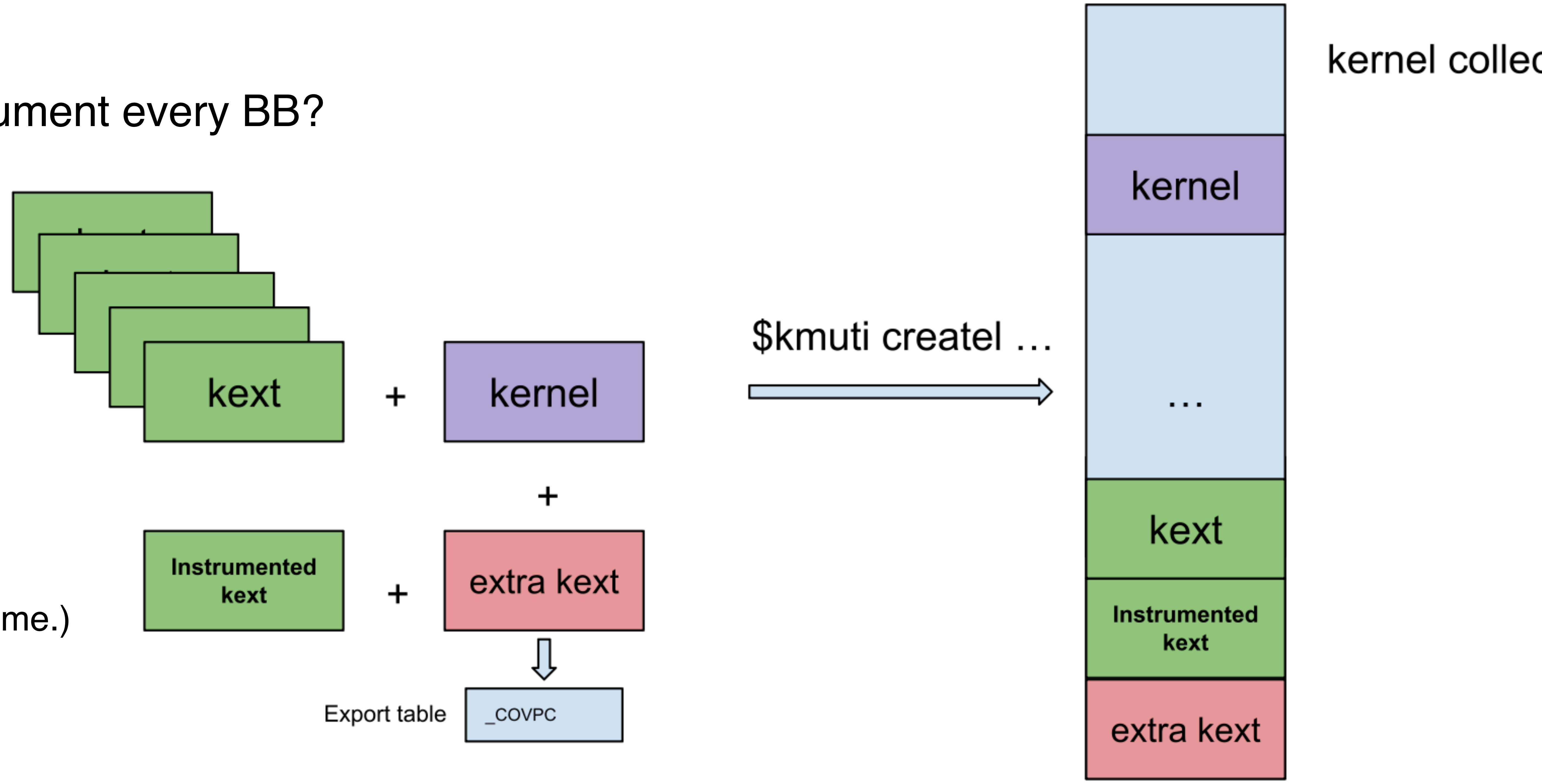


# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

how to instrument every BB?




# Opportunities and challenges.

We can embed a KEXT into kernel collection.

But we don't know the load address.
(The address of _COVPC or any shellcode is unknown at the instrumentation time.)

also

We can't just call into a exported function from arbitrary address. ⬇️
(Calling into a imported function is through __**auth_stubs** and it's clobbering X16, X17 and current LR.)

Other instruction can't be removed without changing intended behavior.

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

how to instrument every BB?

Calling into a imported function is through __**auth_stubs** and it's clobbering X16, X17 and current LR.

```
00000000000360d8 <__auth_stubs>:
   360d8: d0000031      adrp    x17, 0x3c000 <_zalloc_flags+0x3c000>
   360dc: 91000231      add x17, x17, #0x0
   360e0: f9400230      ldr x16, [x17]
   360e4: d71f0a11      braa    x16, x17
```

```
autda   x16, x17
mov     x17, x16
xpacd   x17      ; <<<<----- instrumented and replaced by a call
cmp     x16, x17
b.eq    0x9720
brk     #0xc472
ldar    x9, [x16] ;<<<<----- dereferencing x16.
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

how to instrument every BB?

This needs to patch 5 instructions. To save and restore CPU context, otherwise registers will be clobbered.

```
stp x16, x17, [sp, -16]!   // Push x16 and x17 onto the stack
stp lr, lr, [sp, -16]!     // Push the Link Register (LR) onto the stack
bl COV_                    // Call the COV_ function
ldp lr, lr, [sp], 16       // Pop the Link Register (LR) from the stack
ldp x16, x17, [sp], 16      // Pop x16 and x17 from the stack

COV_:
running original instructions.
ret
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

Problem:

We can’t just put BL instruction into a random address. we need to preserver x16,x17 and LR.

We don’t know where does our KEXT gets loaded to directly jump somewhere in it.

Replacing any instruction with BL needs to patch 5 instructions.

Possible Solution:
what what about modifying __**auth_stubs** or adding a new section to Mach-O?

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

what what about modifying __**auth_stubs** or adding a new section to Mach-O?

kmutil returned an error.
kmutil just ignored added section.

```
else if ( instruction != 0xD503201F ) {
    // ignore imm12 instructions optimized into a NOP, but warn about others
    kcgen_terminate("unknown off12 instruction 0x%08X at 0x%0llX", instruction, fromNewAddress);
}
break;
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

What does kmutil is doing under the hood?

Kmutil is just another binary rewriting tool.

Let's see what has happened to BL and __**auth_stubs** in kernel collection.

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

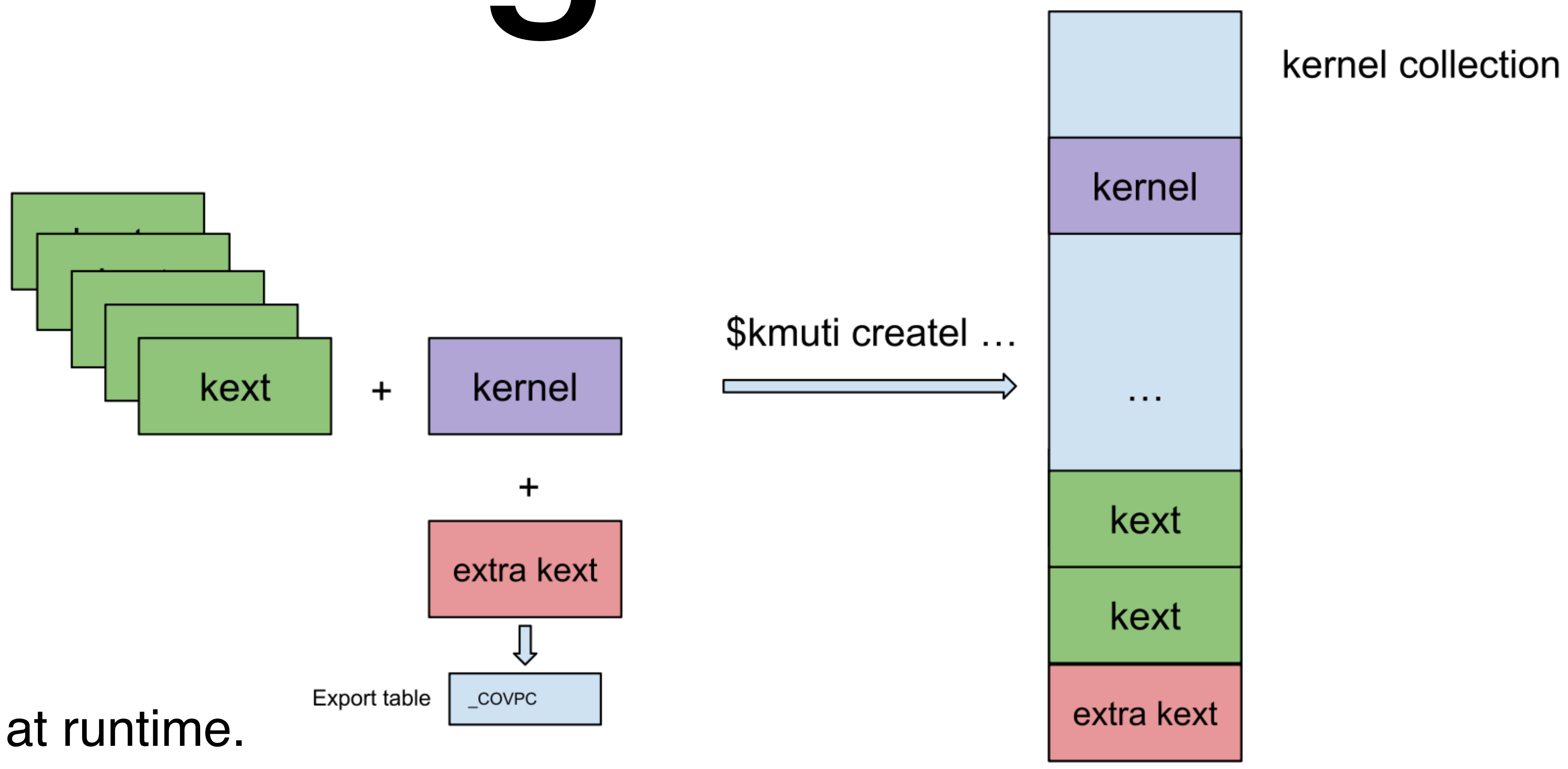
Let's see what has happened to BL and __**auth_stubs** in kernel collection.

kmutil is rewriting KEXT into one blob. Calls are directly to the function, and not through __**auth_stubs**

```
bl to_a_stub_address"  # in your kext will be
# will be
bl fixed_address # in kernel collection.
# they just remove mach-o stub
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:




Static instrumentation:
Binary rewriting

Depreciate: on-demand loadable KEXT, kernel had to bind and fix any relative address at runtime.

Now: prelinked blobs, to speed up the boot process these steps are done at link time when you are creating a Boot Kext Collection.

Not for AKC

In macOS 11 or later Apple has changed its previous scheme of prelinked kernelcaches and Loadable kernel module, to three prelinked kernel collections blobs:

- The Boot Kext Collection (BKC), contains the kernel itself, and all the major system kernel extensions required for a Mac to function.
- The System Kext Collection (SKC), This contains all the other system kernel extensions, which are loaded after booting with the BKC.
- The Auxiliary Kext Collection (AKC), is built and managed by the service kernelmanagerd. This contains all installed third-party kernel extensions, and is loaded after the other two collections.

No clobbering for KEXTs
in Boot Kext collection

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

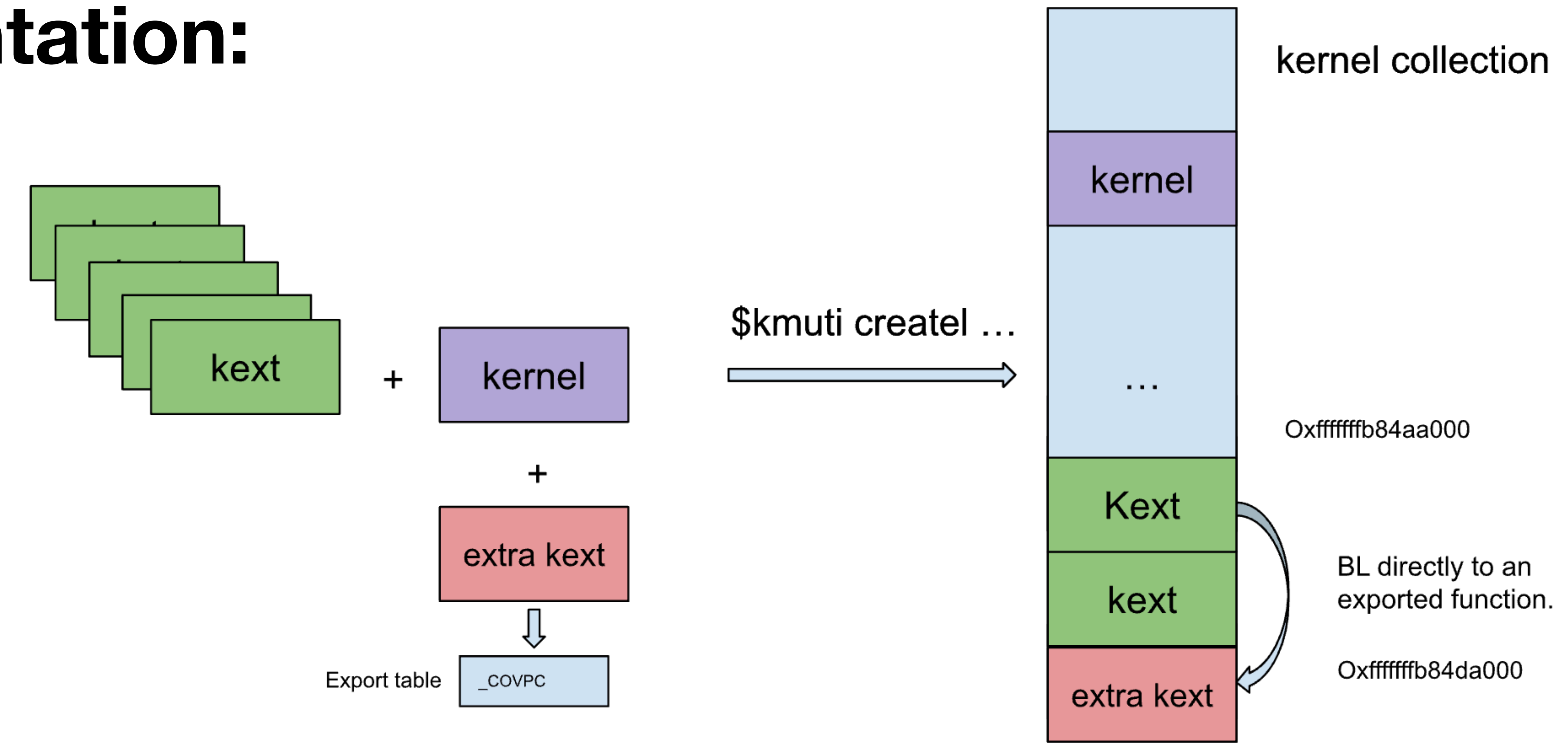
kmutil removes __auth_stubs of boot collection.
this is not a case for AKC. Latter at boot time xnu will load and fix AKC.

```
void AppCacheBuilder::buildAppCache(const std::vector<InputDylib>& dylibs)
 { ....
   if ( removeStubs() ) {
       // Stubs were removed, but we need to rewrite calls which would have gone through those stubs
       rewriteRemovedStubs();
     } else {
   ...
     }
     ...
 }

void AppCacheBuilder::rewriteRemovedStubs()
{
  ...
  // We need to find what the auth stubs pointed to, then rewrite all
  // users of the auth stubs to jump to those locations inst
  ...
}
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting



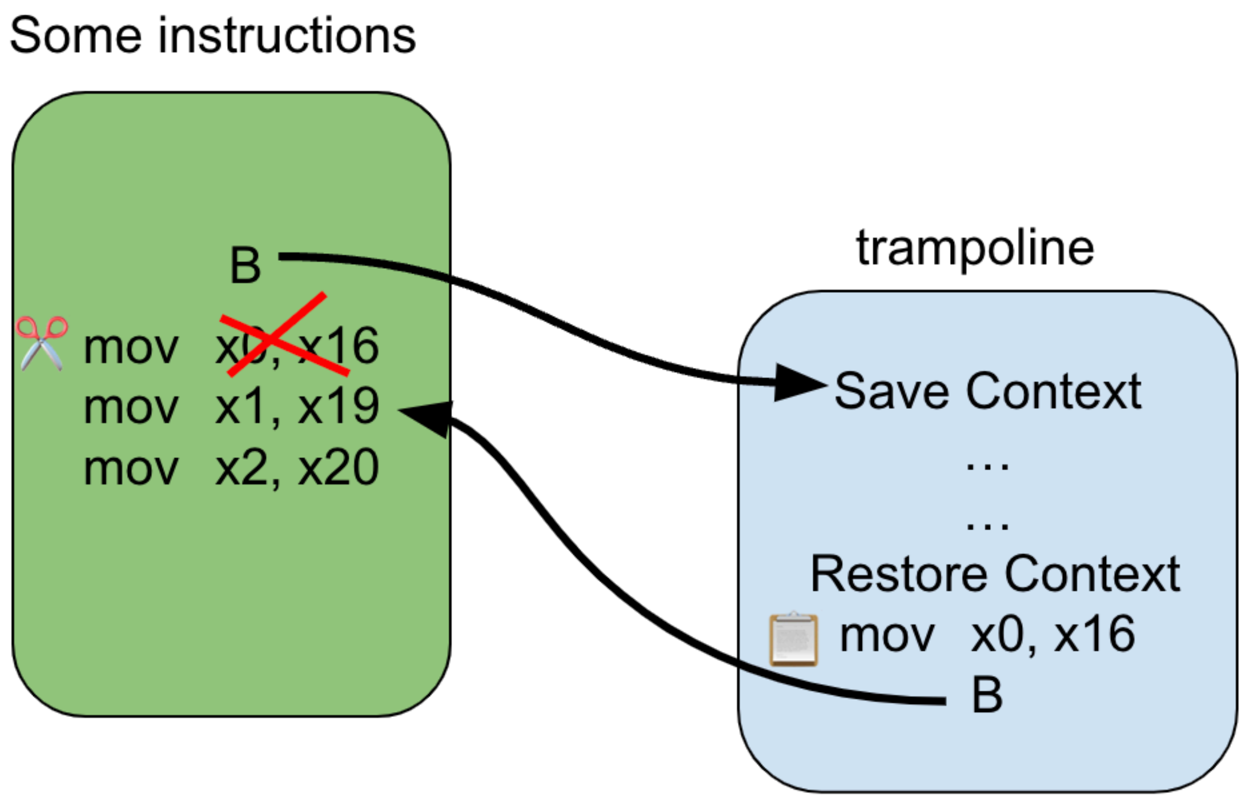

load address of KEXTs within `kernel Collection` are relatives to each other.
It's now one big blob

**consequently** unlink KextFuzz, instead of instrumenting a KEXT's Mach-O file, we instrument them later inside `Boot Kext Collection` blob.

```
bl to_a_stub_address"  # in your kext will be
# will be
bl fixed_address # in kernel collection.
# they just remove mach-o stub
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:




```
void instrument_thunks()
{
    asm volatile (
                ".rept " xstr(REPEAT_COUNT_THUNK) "\n"  // Repeat the following block many times
                "    STR x30, [sp, #-16]!\n"            // save LR. we can't restore it in pop_regs. as we have jumped here.
                "    bl _push_regs\n"
                "    mov x0, #0x0000\n"                 // placeholder targeted_kext flag.
                "    mov x1, #0x4141\n"                 // fix the correct numner when instrumenting as arg0.
                "    mov x1, #0x4141\n"                 // placeholder for BB address
                "    mov x1, #0x4141\n"
                "    mov x1, #0x4141\n"
                "    bl _sanitizer_cov_trace_pc\n"
                "    bl _pop_regs\n"
                "    LDR x30, [sp], #16\n"              // restore LR
                "    nop\n"                             // placeholder for original inst.
                "    nop\n"                             // placeholder for jump back
                ".endr\n"                               // End of repetition
                );
}
```

We can simply put lots of trampolines into our kext.

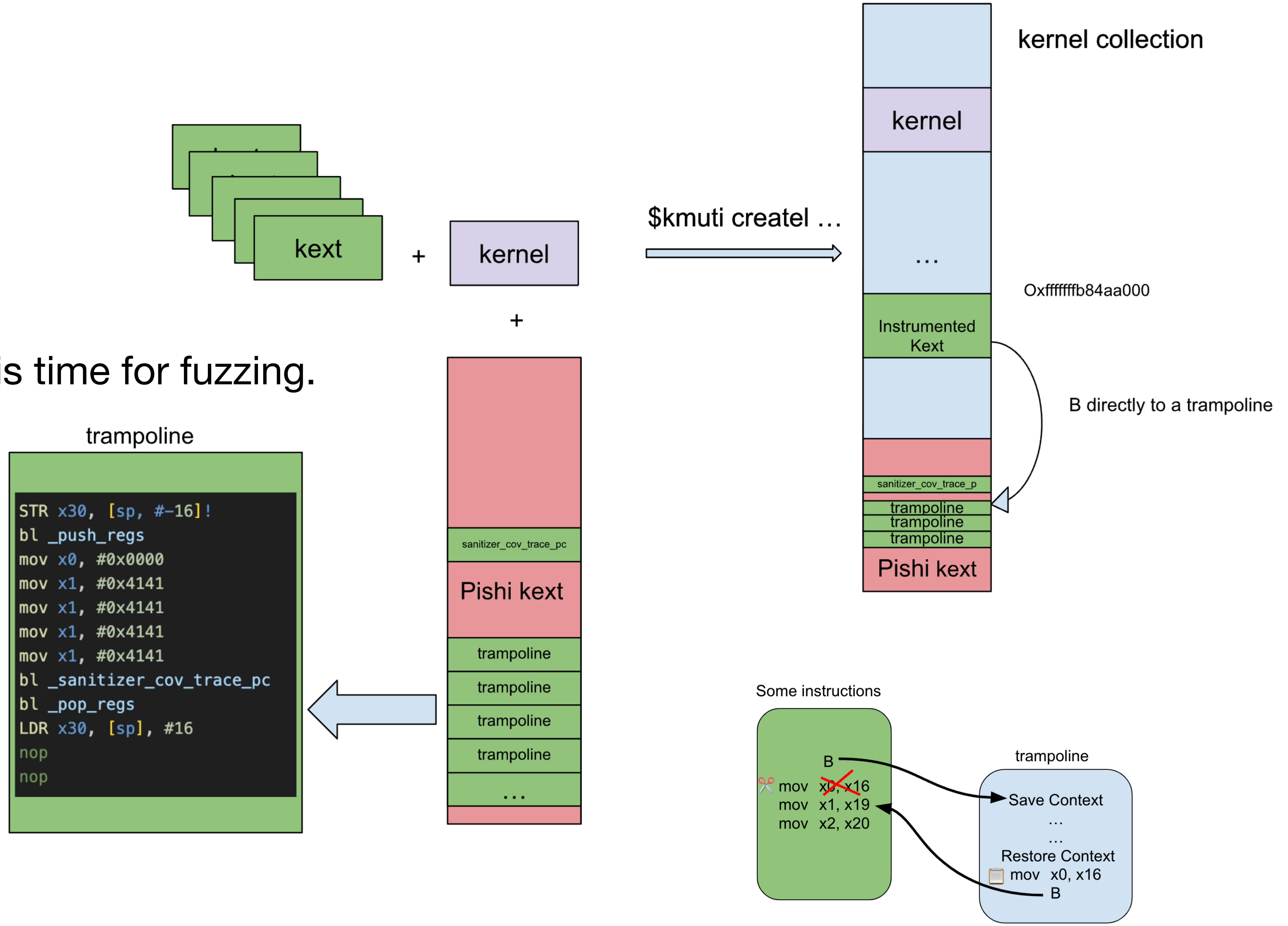
Each Instruction hooking needs its own trampoline,
to be able to execute the original patched instruction.

# KEXT/XNU Fuzzing

Static instrumentation:
Binary rewriting

Memory spray but this time for fuzzing.

```
.rept 0x1000
 nop
.endr
```



# KEXT/XNU Fuzzing

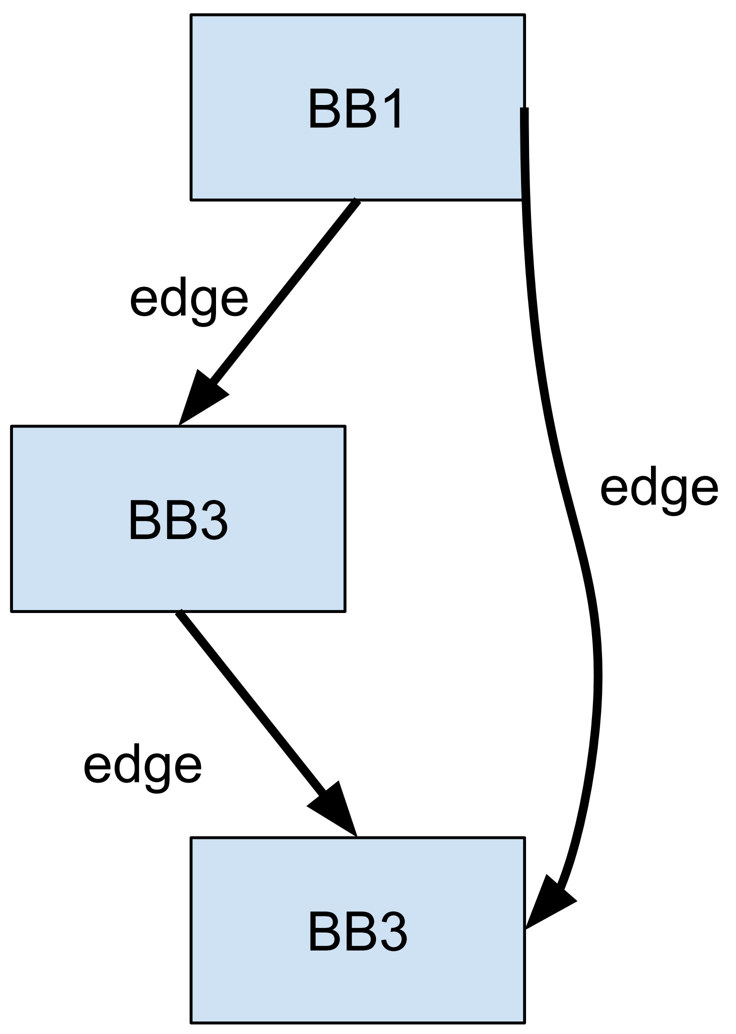
## Software based binary Instrumentation:

What to instrument?

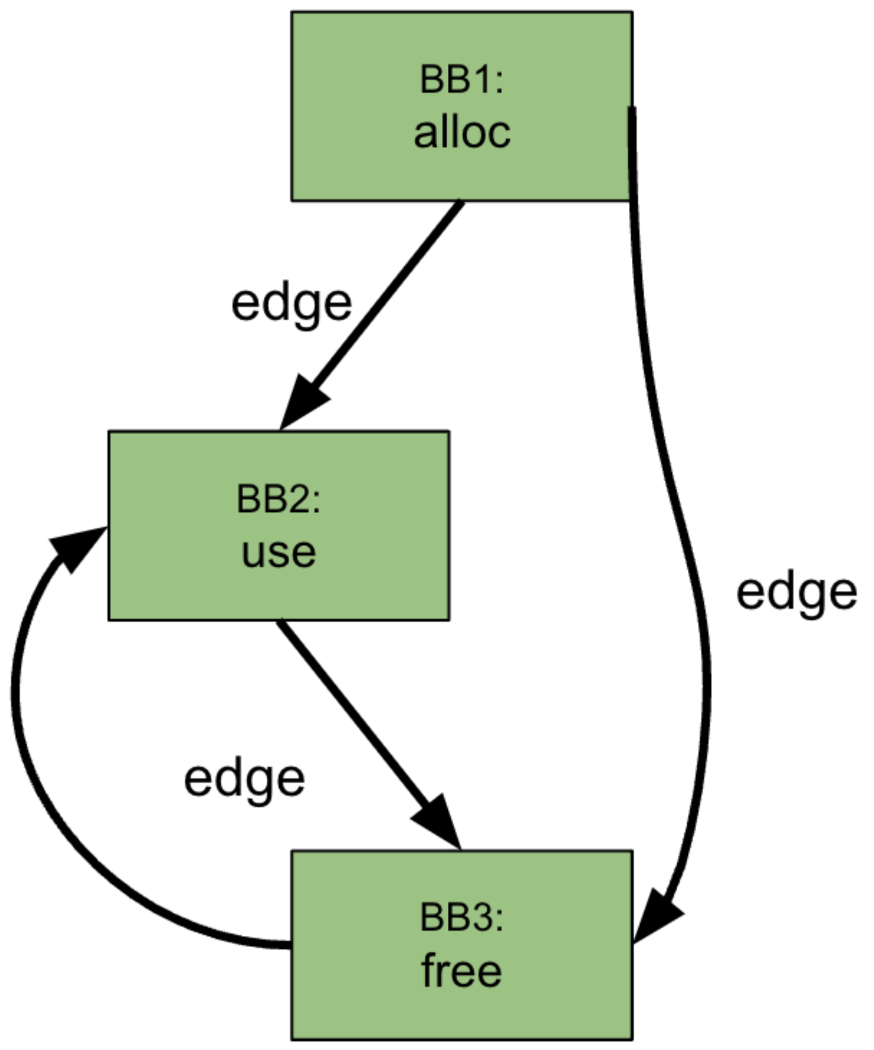
coverage guided fuzzing needs metrics

Fuzzer-centric code coverage metrics:

1. Control flow coverage:
   - Basic blocks coverage
   - Edge coverage
   - Paths coverage
   - Stack Coverage!
2. Data flow coverage




In CFG a Node is BB.




No vulnerability with 100% coverage.

"**control-flow graph** (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution." (Wikipedia)

"a **basic block** is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit." (Wikipedia)

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

BBs are sufficient.

Binary rewriting is difficult.

Even more difficult in the kernel.

Every mistake is panic.

How to instrument?

How to assemble/disassemble?

How to fix the relative instruction?

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Static instrumentation:
Binary rewriting

How to instrument?

After playing with Keystone And thinking about IDA-PRO.
I decide to use Ghidra.

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

How to instrument and how to fix the relative instructions?

Do we even need that?

1. All ARM64 Instructions are 32 bits long.
2. BBs are Disjoint sets ( I explain this later).
3. Almost all ARM64 instructions are non-relative.

following instructions are relative instructions:

- B and its sub instructions are PC relative
- ADR: PC-relative address.
- ADRP: PC-relative address to 4KB page. ( but it definitely has one "add" after it.)
- LDR (literal): Load Register (literal). only one Addressing modes.
- LDRSW (literal): Load Register Signed Word (literal).
- PRFM (literal): Prefetch Memory (literal).

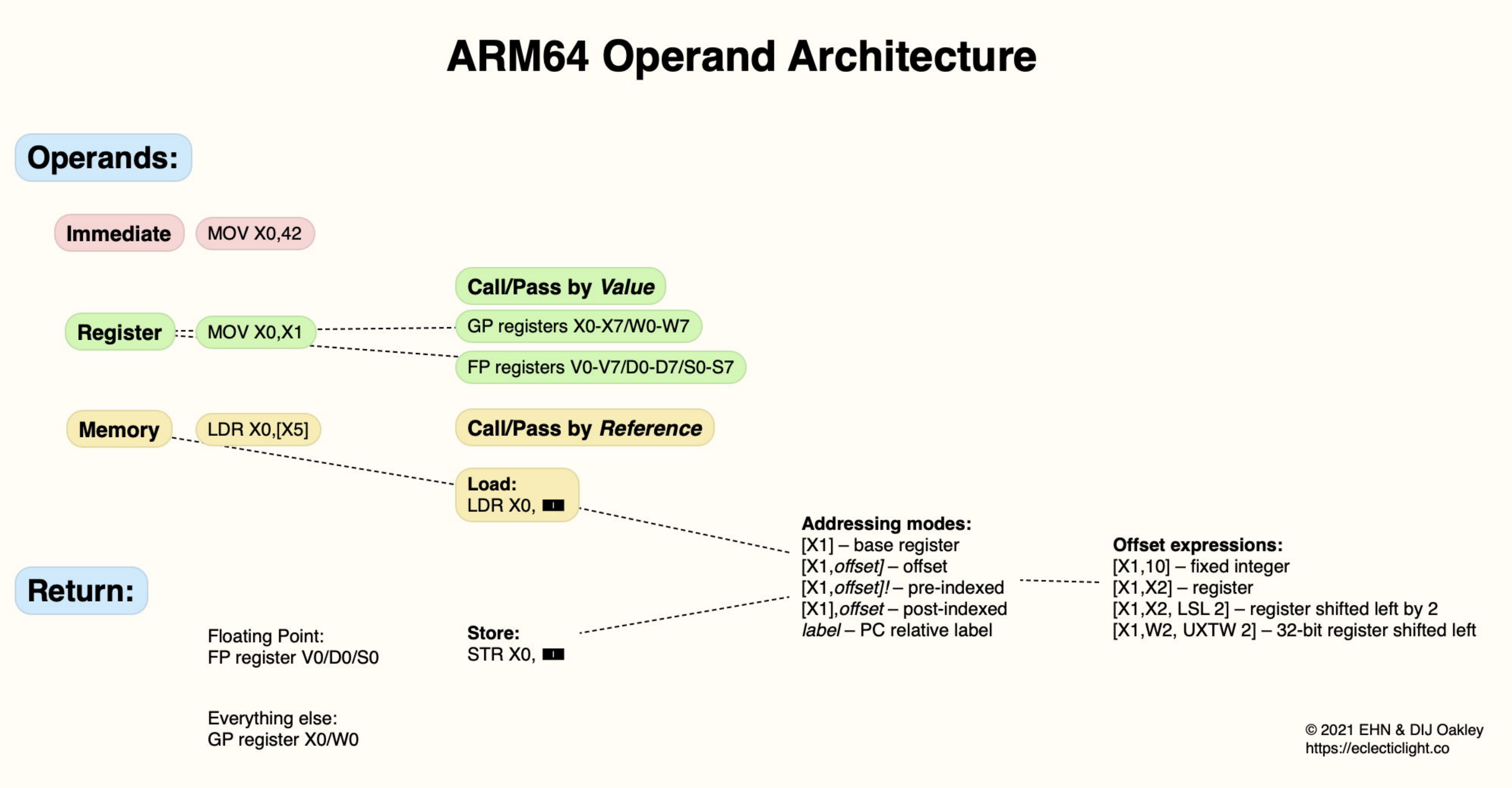
Branches are the edges of the CFG, so they are not part of BBs.

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

But how to fix the relative instructions?

Do we even need that?




AArch64 mnemonics can have 3 types of operands. Immediate, Register, Memory

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

But how to fix the relative instructions?

Do we even need that?

Data movement, arithmetic, logical, shift and rotate, etc instruction.

Almost all ARM64 instructions are non-relative.

We can find at least one
non relative( to current address) instruction inside each BB.

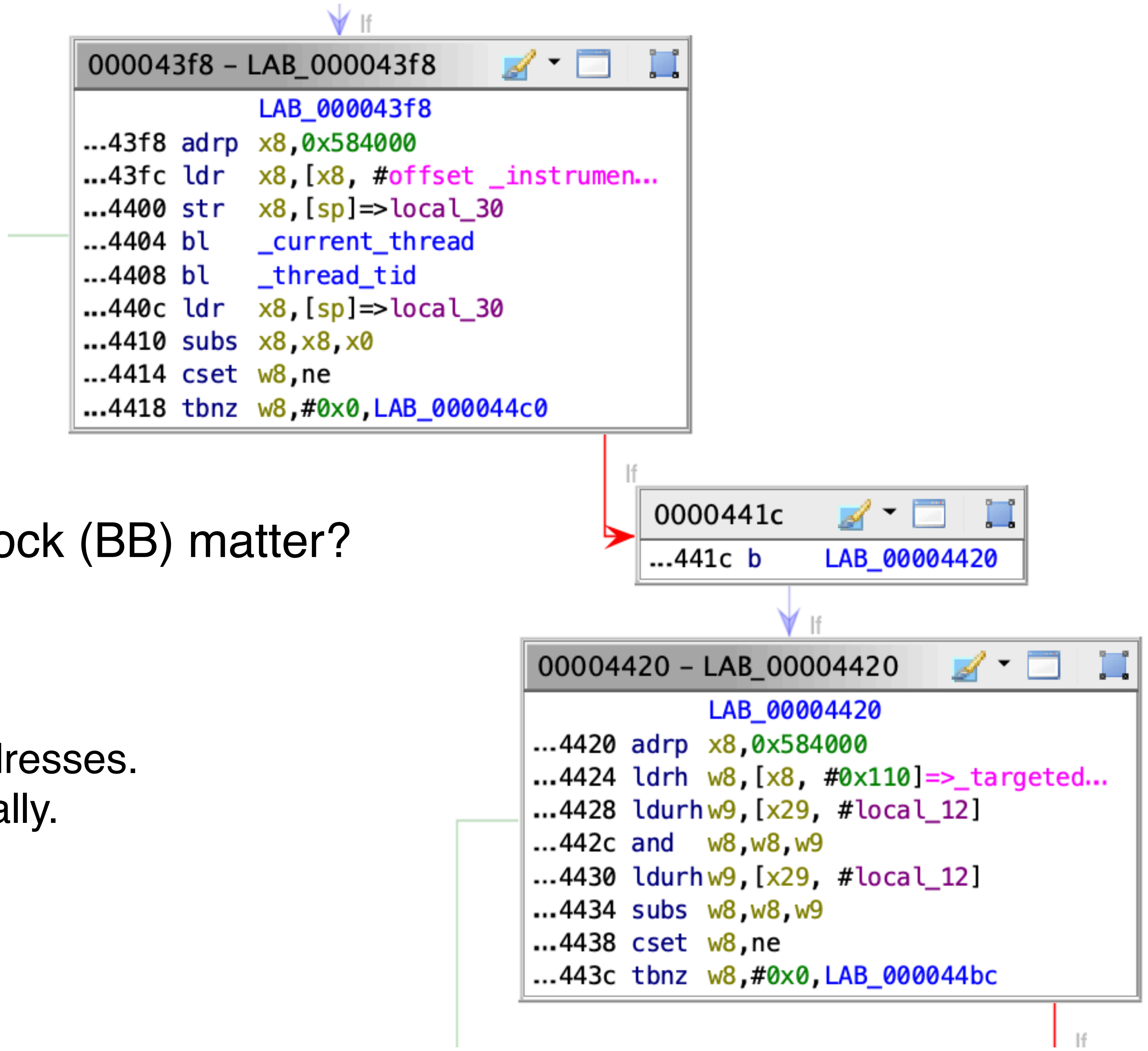
```
Instruction = [
        'and',  'ldadd',  'stur',  'mov',
        'add',  'str',    'ldp',   'bfxil'
        'stp',  'mul',    'lsl',   'sub',
        'lsr',  'cmp',    'tst',   'ldur',
        'orn',  'bic',    'cmn',   'eon',
        'neg',  'adc',    'mvn',   'ana',
        'eor',  'sbc',    'orr',   'ldset',
        'ubfx', 'msub',   'udiv',  'cmhs',
        'xtn',  'fmov',   'sxtw',  'ccmp',
        'asr',  'strb',   'sbfx',  'bfi',
        'strh', 'xtn',    'uxtn',  'sxtw',
        'sxtb', 'sxth',   'uxth',  'uxtb'
        ]
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

But how to fix the relative instructions?

Do we even need that?




Does the location of instrumentation within each basic block (BB) matter?

When doing BB level instrumentation.

No BBs are Disjoint sets ( I explain this later) of addresses.
each instruction of a BB can represent that BB equally.

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

But how to fix the relative instructions?

Do we even need that?

1- Location of instrumentation within each basic block does not matter

2- With high probability there is at least one non relative instruction in every BB.

```
000043f8 - LAB_000043f8
                     LAB_000043f8
...43f8 adrp   x8,0x584000
...43fc ldr    x8,[x8, #offset _instrumen...
...4400 str    x8,[sp]=>local_30
...4404 bl     _current_thread
...4408 bl     _thread_tid
...440c ldr    x8,[sp]=>local_30
...4410 subs   x8,x8,x0
...4414 cset   w8,ne
...4418 tbnz   w8,#0x0,LAB_000044c0
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

We can enumerate BBs in Ghidra.
We can disassemble/assemble instructions.
you can find at least one instruction in every basic block (BB) that is non PC-relative.

```
all_basic_blocks = get_basic_blocks(toAddr(start_address), toAddr(end_address)) # type: i
if not all_basic_blocks:
    print("all_basic_blocks is empty check if start_address and end_address is correct.")
```

Ghidra script:
find stubs in our KEXT.
find BBs in requested address ranges.
loop into BBs: find one non-relative instruction.
replace it with jump to stub.
rewrite the stub: use next stub.

```
assembler = Assemblers.getAssembler(currentProgram) # type: ignore
create_label(stub_address, "meysam_stub_number_" + str(bb_index))
create_label(patch_address.add(INSTRUCTION_SIZE), "meysam_return_number_" + str(bb_index))

# Patch the BB to jump to out stub_address# label
patched_instruction = "b {}".format("meysam_stub_number_" + str(bb_index)) # Change this to your desired instruction
assemble_opcode(assembler, patch_address, patched_instruction)
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

We can enumerate BBs in Ghidra.
We can disassemble/assemble instructions.
you can find at least one instruction in every basic block (BB) that is non PC-relative.

Ghidra script:
find stubs in our KEXT.
find BBs in requested address ranges.
loop into BBs: find one non-relative instruction.
replace it with jump to stub.
rewrite the stub: use next stub

Before: Target BB

```
                        FUN_fffffe000a800d90
...e000a800d90 7f 23 03 d5    pacibsp
...e000a800d94 ff c3 05 d1    sub        sp,sp,#0x170
...e000a800d98 fc 6f 15 a9    stp        x28,x27,[sp, #0x150]
...e000a800d9c fd 7b 16 a9    stp        x29,x30,[sp, #0x160]
...e000a800da0 fd 83 05 91    add        x29,sp,#0x160
...e000a800da4 48 a9 fe 90    adrp       x8,-0x1fff82d8000
...e000a800da8 08 15 40 f9    ldr        x8,[x8, #0x28]
...e000a800dac 08 01 40 f9    ldr        x8,[x8]
...e000a800db0 a8 83 1e f8    stur       x8,[x29, #-0x18]
```

Before: Stubs

```
...e000a28298c fe 0f 1f f8    str     x30,[sp, #-0x10]!
...e000a282990 34 ff ff 97    bl      _push_regs
...e000a282994 00 00 80 d2    mov     x0,#0x0
...e000a282998 21 28 88 d2    mov     x1,#0x4141
...e000a28299c 21 28 88 d2    mov     x1,#0x4141
...e000a2829a0 21 28 88 d2    mov     x1,#0x4141
...e000a2829a4 21 28 88 d2    mov     x1,#0x4141
...e000a2829a8 76 ff ff 97    bl      _sanitizer_cov_trace_pc
...e000a2829ac 51 ff ff 97    bl      _pop_regs
...e000a2829b0 fe 07 41 f8    ldr     x30,[sp], #0x10
...e000a2829b4 1f 20 03 d5    nop
...e000a2829b8 1f 20 03 d5    nop
...e000a2829bc fe 0f 1f f8    str     x30,[sp, #-0x10]!
...e000a2829c0 28 ff ff 97    bl      _push_regs
...e000a2829c4 00 00 80 d2    mov     x0,#0x0
...e000a2829c8 21 28 88 d2    mov     x1,#0x4141
...e000a2829cc 21 28 88 d2    mov     x1,#0x4141
...e000a2829d0 21 28 88 d2    mov     x1,#0x4141
...e000a2829d4 21 28 88 d2    mov     x1,#0x4141
...e000a2829d8 6a ff ff 97    bl      _sanitizer_cov_trace_pc
...e000a2829dc 45 ff ff 97    bl      _pop_regs
...e000a2829e0 fe 07 41 f8    ldr     x30,[sp], #0x10
...e000a2829e4 1f 20 03 d5    nop
...e000a2829e8 1f 20 03 d5    nop
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

We can enumerate BBs in Ghidra.
We can disassemble/assemble instructions.
you can find at least one instruction in every basic block (BB) that is non PC-relative.

Ghidra script:
find stubs in our KEXT.
find BBs in requested address ranges.
loop into BBs: find one non-relative instruction.
replace it with jump to stub.
rewrite the stub: use next stub

After: Target BB

```
                        FUN_fffffe000a800d90
...e000a800d90 7f 23 03 d5     pacibsp
...e000a800d94 fe 06 ea 17     b          meysam_stub_number_0

                        meysam_return_number_0
...e000a800d98 fc 6f 15 a9     stp        x28,x27,[sp, #0x150]
...e000a800d9c fd 7b 16 a9     stp        x29,x30,[sp, #0x160]
...e000a800da0 fd 83 05 91     add        x29,sp,#0x160
...e000a800da4 48 a9 fe 90     adrp       x8,-0x1fff82d8000
...e000a800da8 08 15 40 f9     ldr        x8,[x8, #0x28]
```

After: Stubs

```
                        meysam_stub_number_0
e000a28298c fe 0f 1f f8     str        x30,[sp, #-0x10]!
e000a282990 34 ff ff 97     bl         _push_regs
e000a282994 20 00 80 d2     mov        x0,#0x1
e000a282998 81 b2 81 d2     mov        x1,#0xd94
e000a28299c 01 50 a1 f2     movk       x1,#0xa80, LSL #16
e000a2829a0 01 c0 df f2     movk       x1,#0xfe00, LSL #32
e000a2829a4 e1 ff ff f2     movk       x1,#0xffff, LSL #48
e000a2829a8 76 ff ff 97     bl         _sanitizer_cov_trace_pc
e000a2829ac 51 ff ff 97     bl         _pop_regs
e000a2829b0 fe 07 41 f8     ldr        x30,[sp], #0x10
e000a2829b4 ff c3 05 d1     sub        sp,sp,#0x170
e000a2829b8 f8 f8 15 14     b          meysam_return_number_0
```

# KEXT/XNU Fuzzing

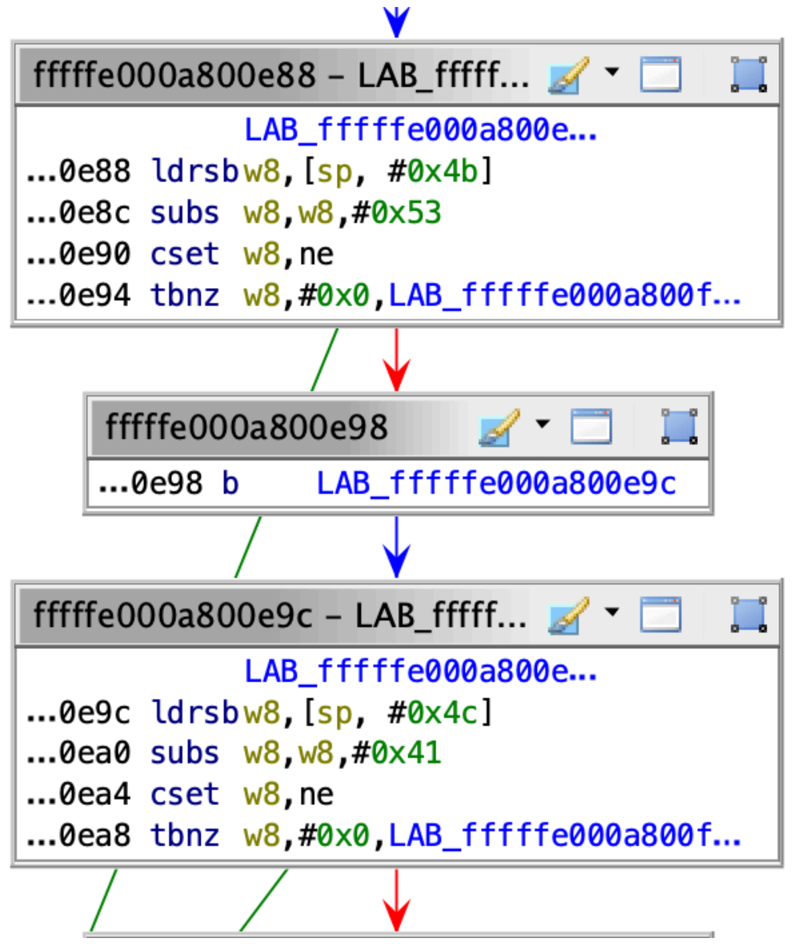
## Software based binary Instrumentation:

We can enumerate BBs in Ghidra.
We can disassemble/assemble instructions.

Coverage efficiency

The majority of the basic blocks we didn't instrument consist of only a single B instruction.
more instruction can be instrumented.

99.72% of valuable BBs. BBs with at least one data movement, arithmetic, logical, shift and rotate, etc instruction.




```
                    thunk_FUN_fffffe00083e7e08

009755fe8 5f 24 03 d5      bti          c
009755fec 87 47 b2 17      b            FUN_fffffe00083e7e08
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Collect all coverages only for fuzzer thread

share it over shared memory with the fuzzer.

```c
void sanitizer_cov_trace_pc(uint16_t kext, uintptr_t address)
{
    if ( __improbable(do_instrument) ) {
        /* number of cases we want to reject due to wrong thread id is a lot more than targeted_kext so we compare it first. */
        if( __improbable(instrumented_thread == thread_tid(current_thread())) ) {
            /*
                I just added targeted_kext to be able to instument multiple KEXTs at once,
                instead of build/install/boot for each KEXT. simple benchmark shows it has not that much performance penalty.
            */
            if ( __probable( (targeted_kext & kext) == kext) ) {
                if ( __improbable(coverage_area == NULL) )
                    return;

                /* The first 64-bit word is the number of subsequent PCs. */
                if ( __probable(coverage_area->kcov_pos < 0x20000) ) {
                    unsigned long pos = coverage_area->kcov_pos;
                    coverage_area->kcov_area[pos] = address;
                    coverage_area->kcov_pos +=1;
                }
            }
        }
    }
}
```


docs.kernel.org/dev-tools/kcov.html

coverage for fuzzing

lection

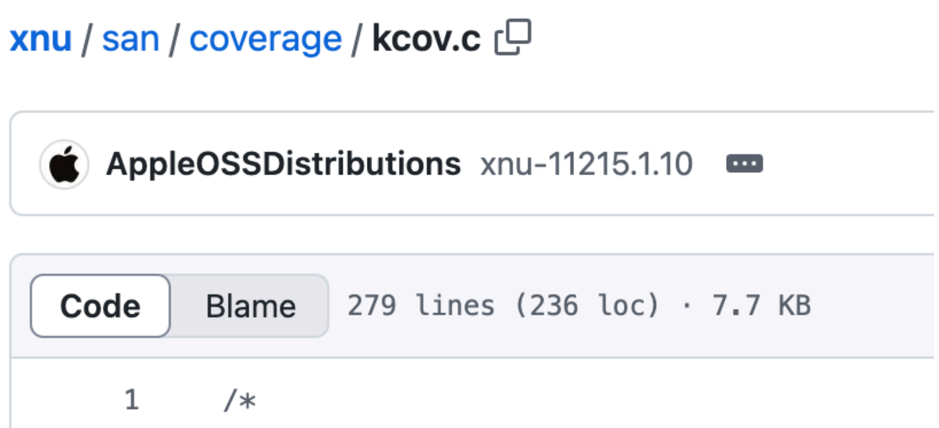
operands collection

rage collection

KCOV: code coverage for fuzzing

KCOV collects and exposes kernel code coverage information in a

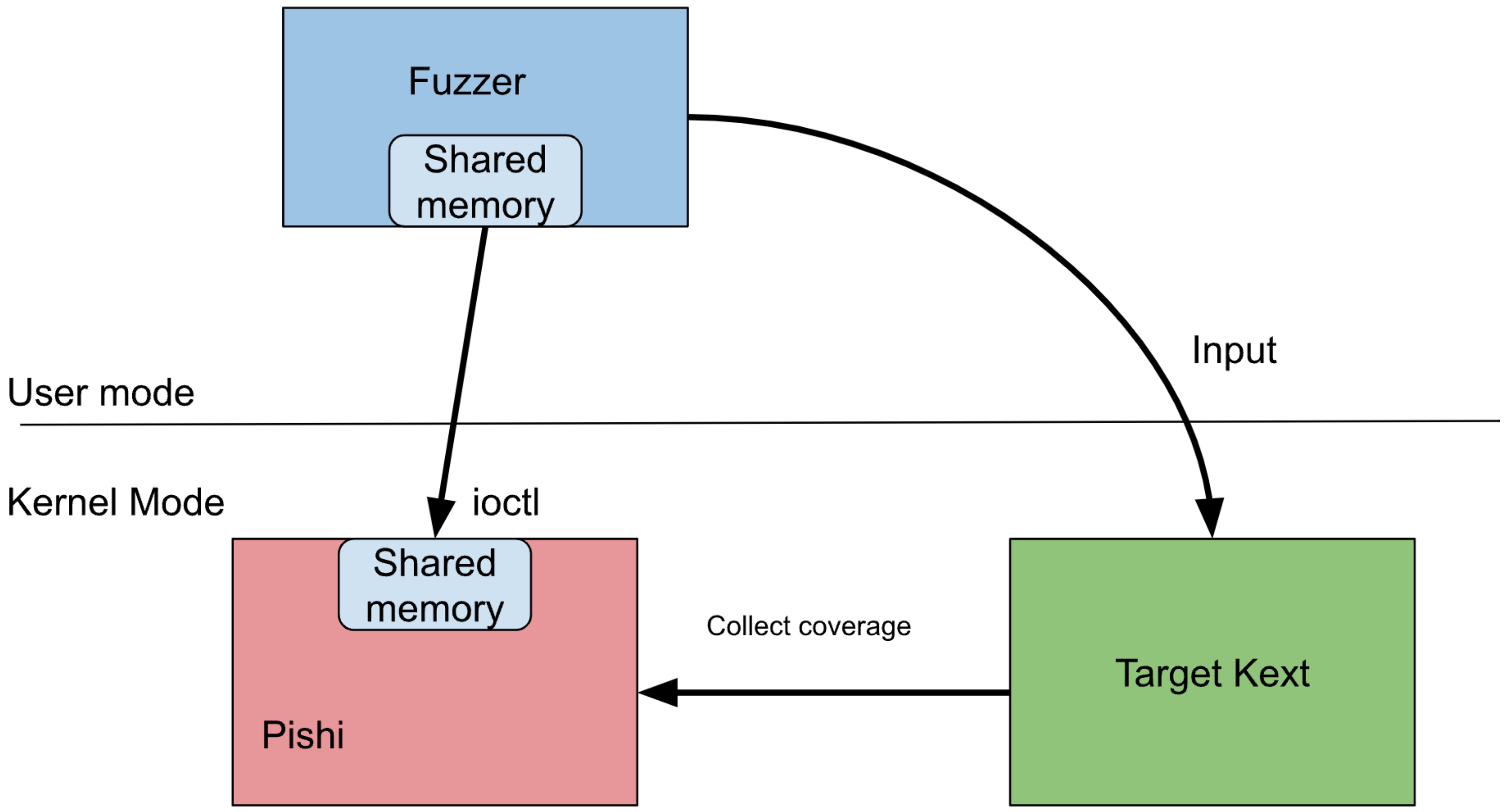
Coverage data of a running kernel is exported via the `kcov` debug


xnu / san / coverage / kcov.c

AppleOSSDistributions xnu-11215.1.10

Code Blame 279 lines (236 loc) · 7.7 KB

1 /*



# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Does it works?
Let’s instrument one sample function.

```
void fuzz_me(uintptr_t* p)
{
    int error = 0;
    size_t len;
    char k_buffer[0x100] = {0};
    error = copyinstr((user_addr_t)*p, k_buffer, sizeof(k_buffer), &len);
    if ( error ) {
        print_message("[PISHI] can't copyinstr\n");
        return;
    }

    if ( strlen(k_buffer) > 9 )
        if( k_buffer[0] =='M' )
            if( k_buffer[1] =='E' )
                if( k_buffer[2] =='Y' )
                    if( k_buffer[3] =='S' )
                        if( k_buffer[4] =='A' )
                            if( k_buffer[5] =='M' )
                                if( k_buffer[6] =='6' )
                                    if( k_buffer[7] =='7' )
                                        if( k_buffer[8] =='8' )
                                            if( k_buffer[9] =='9' ) {
                                                printf("boom!\n");
                                                int* p = (int*)0x41414141;
                                                *p = 0x42424242;
                    }
}
```

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Does it works?
Let’s instrument one sample function.

But how to feed the coverage to a fuzzer?

I have used extra counters before in libFuzzer to feed additional coverage.

```
void cover_stop()
{
        uint64_t ncov = __atomic_load_n(&kcov_data[0], __ATOMIC_RELAXED);
        if (ncov >= KCOV_COVER_SIZE)
                fail("too much cover: %llu", ncov);
        for (uint64_t i = 0; i < ncov; i++) {
                uint64_t pc = __atomic_load_n(&kcov_data[i + 1], __ATOMIC_RELAXED);
                libfuzzer_coverage[pc % sizeof(libfuzzer_coverage)]++;
        }
}
```

kcovfuzzer.c

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Does it works?
Let's instrument one sample function.

But how to feed the coverage to a fuzzer?
once I have used extra counter in libFuzzer to feed extra coverage to it.

```
#if LIBFUZZER_APPLE

namespace fuzzer {
uint8_t *ExtraCountersBegin() { return nullptr; }
uint8_t *ExtraCountersEnd() { return nullptr; }
void ClearExtraCounters() {}
} // namespace fuzzer

#endif
```

FuzzerExtraCountersDarwin.cpp

# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

Does it works?
Let’s instrument one sample function.

No problem:
git clone llvm
git patch
build.

```
 namespace fuzzer {
-uint8_t *ExtraCountersBegin() { return nullptr; }
-uint8_t *ExtraCountersEnd() { return nullptr; }
-void ClearExtraCounters() {}
+extern "C" char _pishi_libfuzzer_coverage[32 << 10];
+
+uint8_t *ExtraCountersBegin() { return (uint8_t *)_pishi_libfuzzer_coverage; }
+uint8_t *ExtraCountersEnd() { return ((uint8_t *) _pishi_libfuzzer_coverage) + sizeof(_pishi_libfuzzer_coverage); }
+
+void ClearExtraCounters()
+{
+    uintptr_t *Beg = reinterpret_cast<uintptr_t*>(ExtraCountersBegin());
+    uintptr_t *End = reinterpret_cast<uintptr_t*>(ExtraCountersEnd());
+    for (; Beg < End; Beg++) {
+    *Beg = 0;
+    __asm__ __volatile__("" : : : "memory");
+  }
+
 } // namespace fuzzer
+}
```

# KEXT/XNU Fuzzing

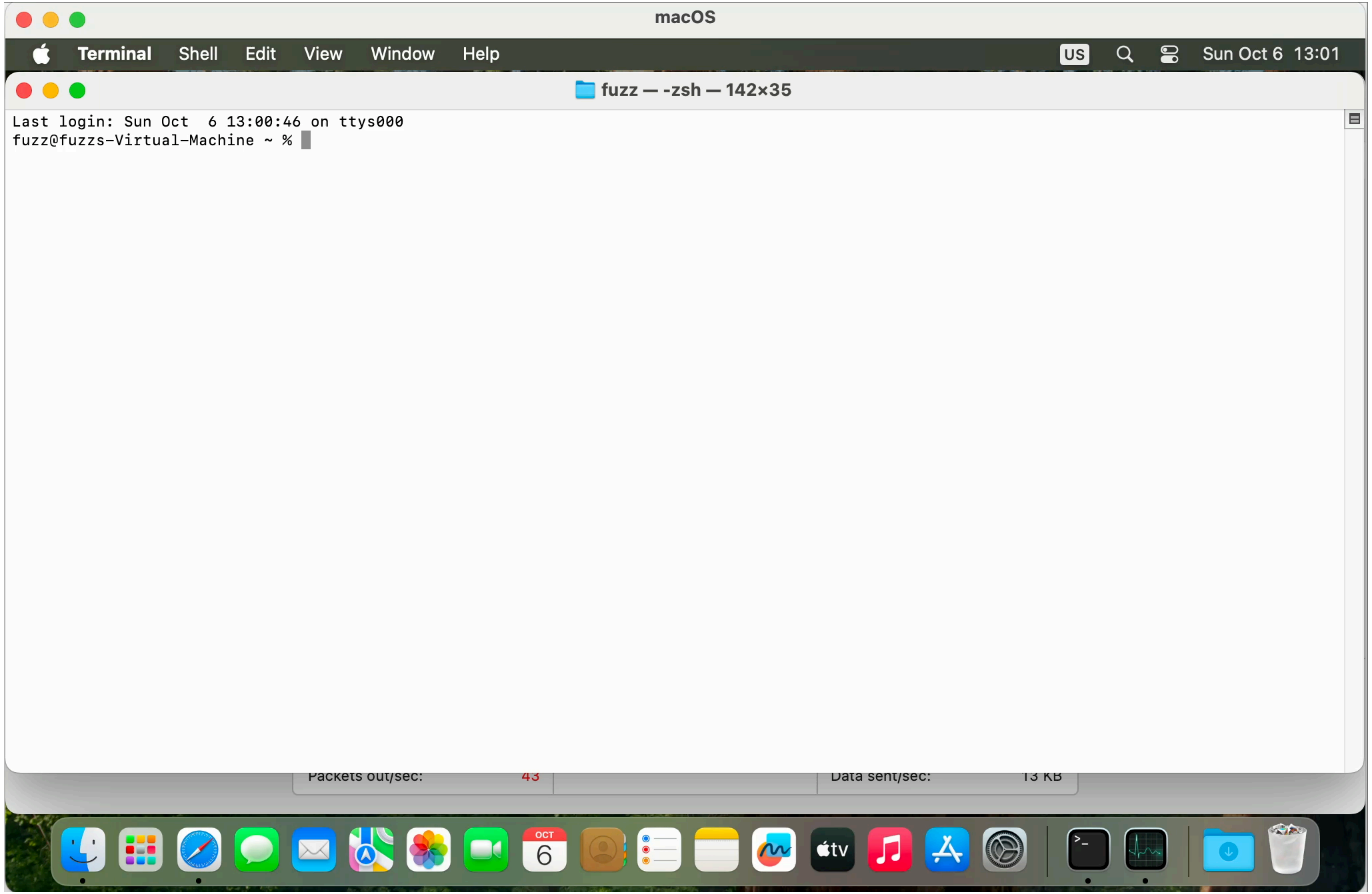
## Software based binary Instrumentation:

Does it works?
Let's instrument one sample function.

Just wait a few seconds. We will get a panic.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {

  pishi_start(CONFIG_JSON);

  uintptr_t **a = (uintptr_t **)&data;
  ioctl(pishi_fd, PISHI_IOCTL_FUZZ, a);

  pishi_stop();

  return 0;
}
```

# KEXT/XNU Fuzzing




Demo

# KEXT/XNU Fuzzing

**Software based binary Instrumentation:**

How to fuzz system calls?

We just fuzzed a function in the kernel with libFuzzer

libprotobuf-mutator, Structure-Aware Fuzzing with libFuzzer

# KEXT/XNU Fuzzing

How did I instrument XNU?

We can’t just instrument all BBs in XNU.

xnu / san / coverage / kcov-blacklist

AppleOSSDistributions xnu-10063.101.15

Code Blame 32 lines (27 loc) · 641 Bytes

```
# Blanket ignore non-sanitized functions
fun:ksancov_*
fun:kcov_*
fun:dtrace_*

# Exclude KSANCOV itself
src:./san/coverage/kcov.c
src:./san/coverage/kcov_ksancov.c
src:./san/coverage/kcov_stksz.c

# Exclude KASan runtime
src:./san/memory/*

src:./osfmk/kern/debug.c

# Calls from sanitizer hook back to kerne
fun:_disable_preemption
fun:_enable_preemption
fun:current_thread
fun:ml_at_interrupt_context
fun:get_interrupt_level
fun:get_active_thread
fun:cpu_datap
fun:cpu_number
fun:get_cpu_number
fun:pmap_in_ppl
fun:get_preemption_level

# Closure of VM_KERNEL_UNSLIDE
fun:vm_memtag_add_ptr_tag
fun:ml_static_unslide
fun:vm_is_addr_slid
```

xnu / san / coverage / kcov-blacklist-arm64

AppleOSSDistributions xnu-10063.101.15

Code Blame 18 lines (15 loc) · 444 Bytes

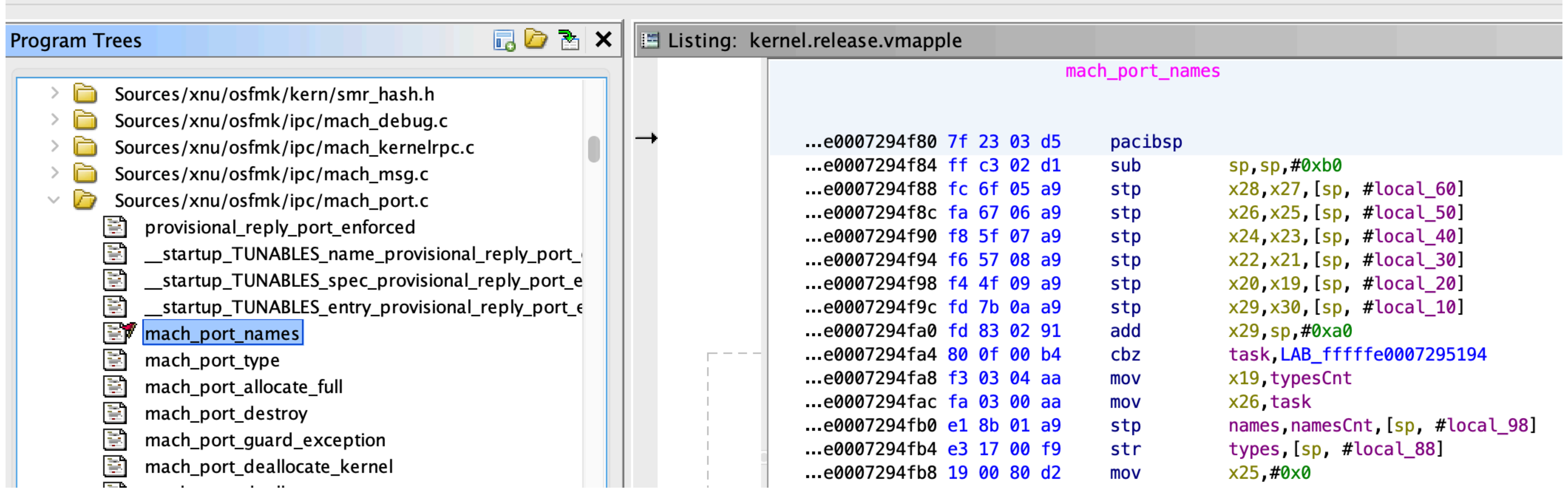
```
# ARM64 specific blacklist

# Exclude KASan runtime
src:./osfmk/arm/machine_routines_common.c

# These use a local variable to work out which st
# a fakestack allocation.
fun:ml_at_interrupt_context
fun:ml_stack_remaining
fun:ml_stack_base
fun:ml_stack_size
fun:kernel_preempt_check

# Closure of pmap_in_ppl
fun:pmap_interrupts_disable
fun:pmap_get_cpu_data
fun:ml_get_ppl_cpu_data
fun:pmap_interrupts_restore
```

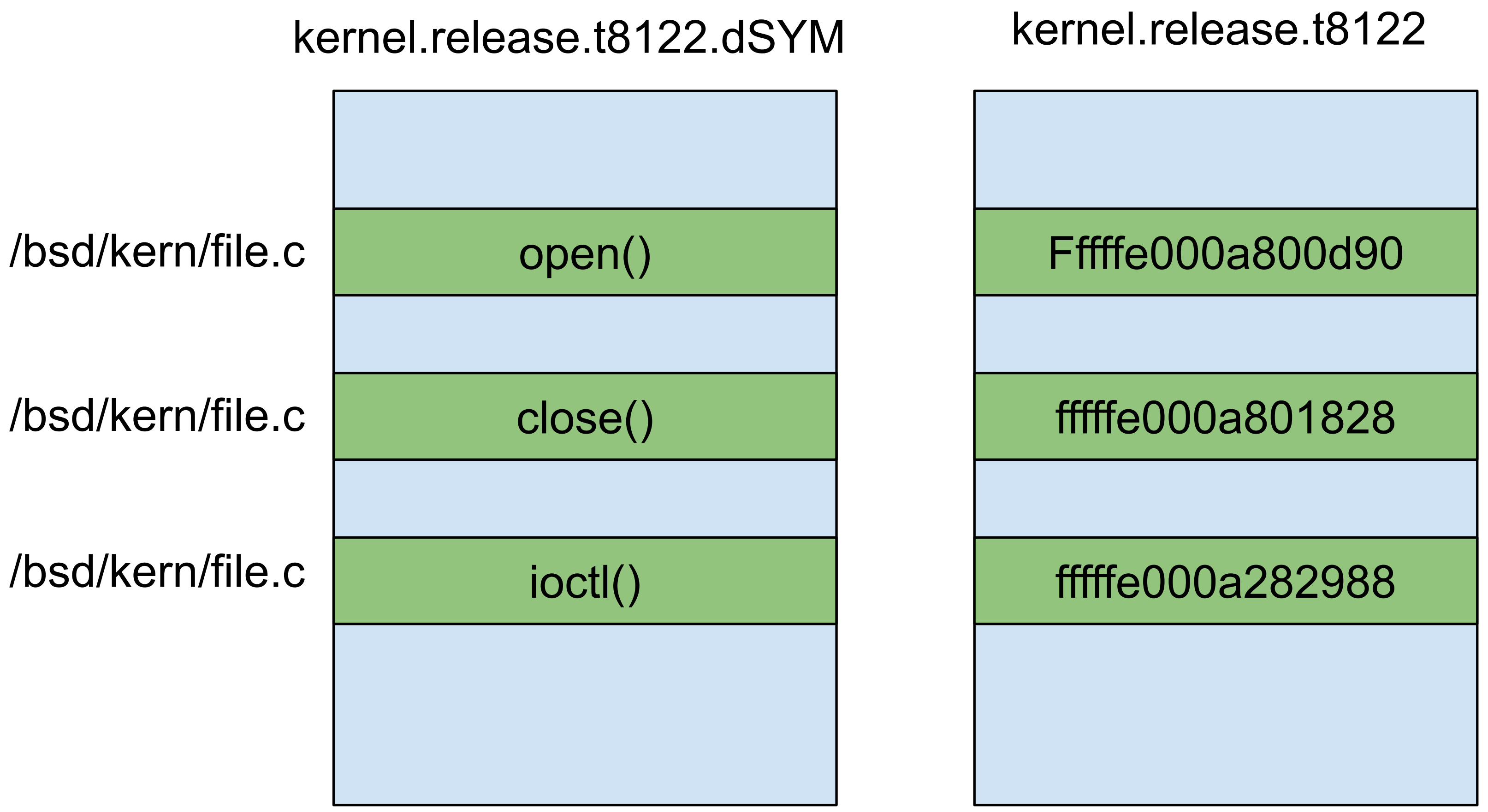
# KEXT/XNU Fuzzing

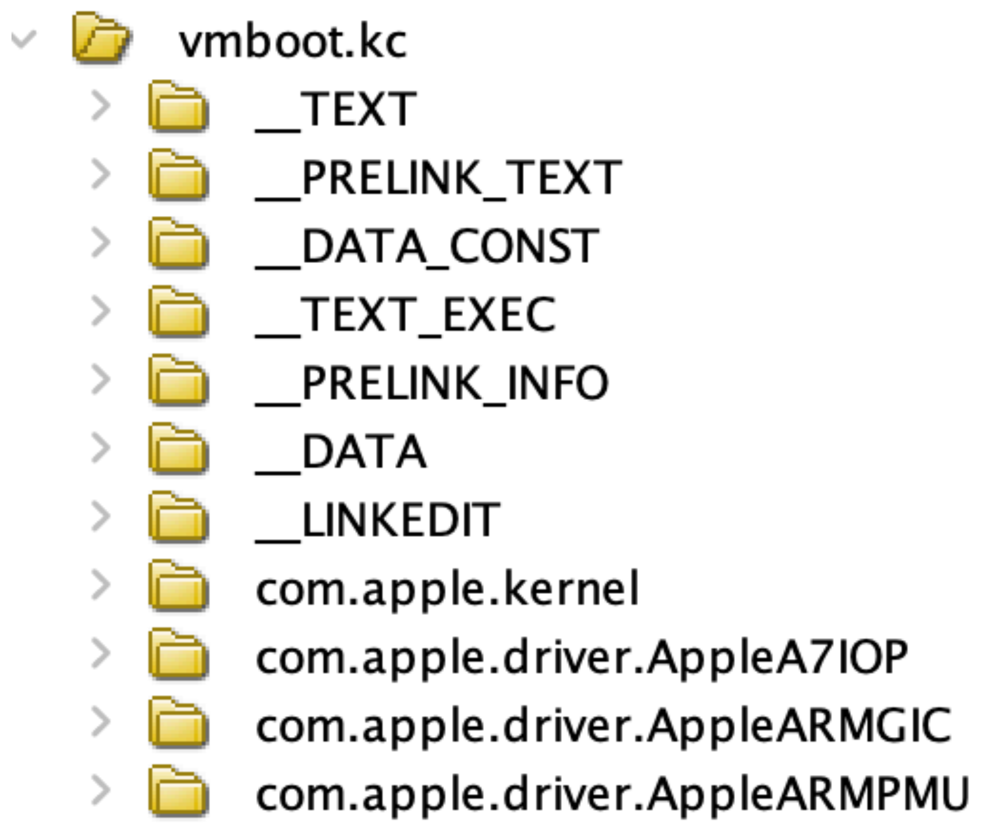
How did I instrument XNU?

We can't just instrument all BBs in XNU.

KDK contains DWARF files.




Functions are at same offset.

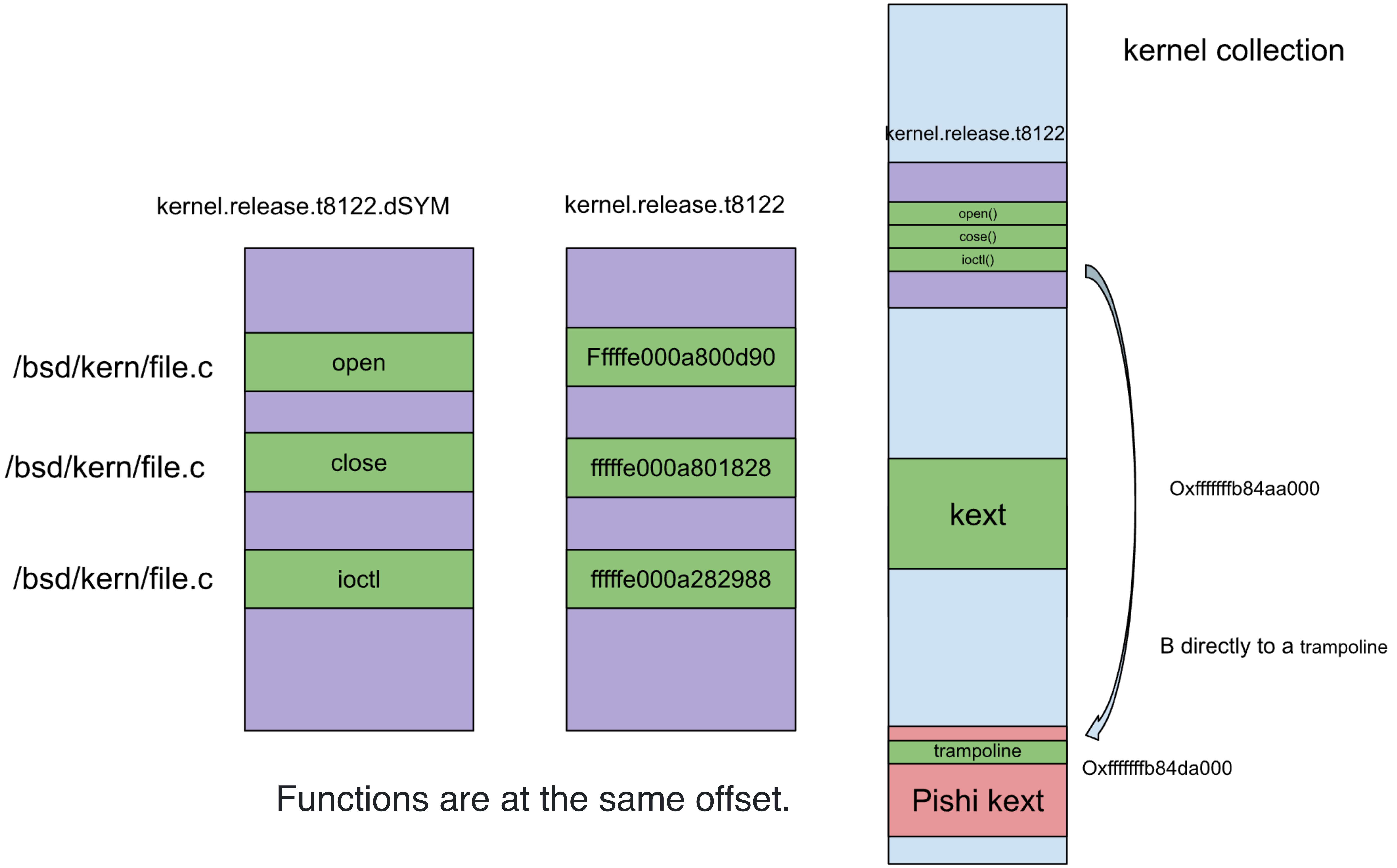
# KEXT/XNU Fuzzing

How did I instrument XNU?

We can't just instrument all BBs in XNU.

Extract offsets from DWARF file
We can filter functions by path and name

Label offsets in kernel collection.



# KEXT/XNU Fuzzing

## Software based binary Instrumentation:

libprotobuf-mutator, Structure-Aware Fuzzing with libFuzzer

libFuzzer can be turned into a grammar-aware (i.e. **structure-aware**) fuzzing engine for a specific input type.

Protobufs provide a convenient way to serialize structured data,
and LPM provides an easy way to mutate protobufs for structure-aware fuzzing.

Pishi is a tool you can hook into another fuzzer e.g. LibAFL

Project Zero

News and updates from the Project Zero team at Google

Thursday, April 22, 2021

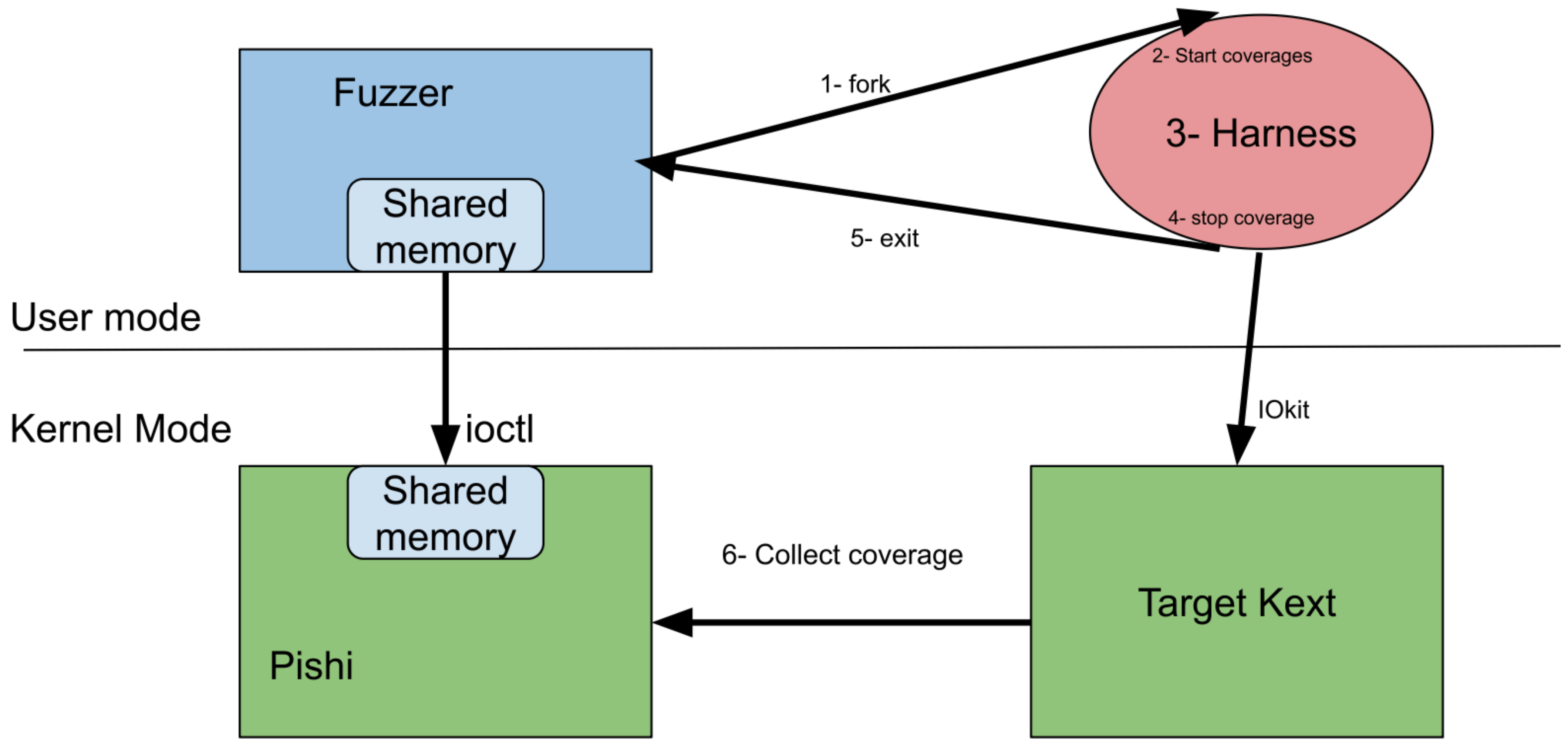
Designing sockfuzzer, a network syscall fuzzer for XNU

Posted by Ned Williamson, Project Zero

# KEXT/XNU Fuzzing

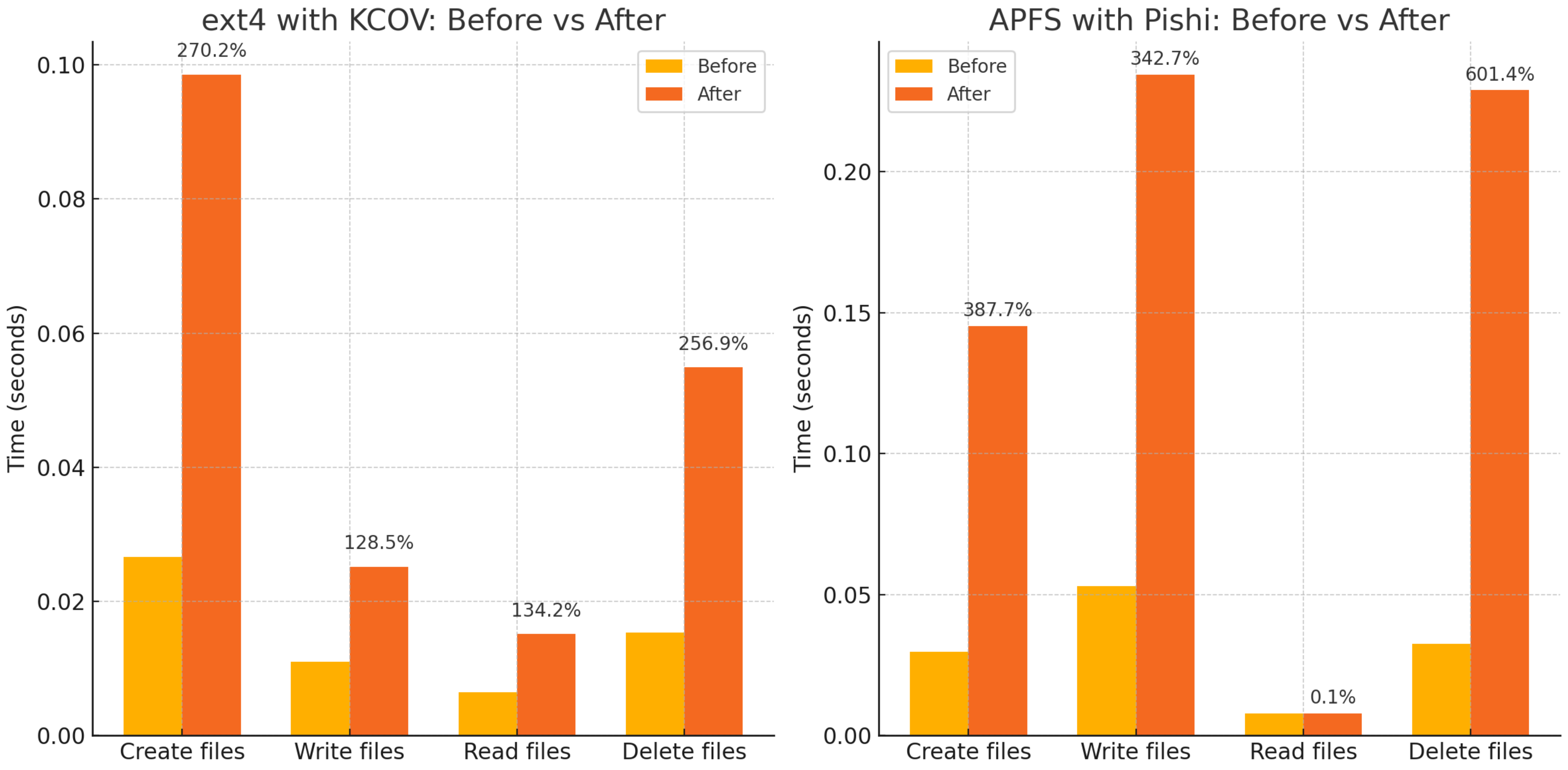
## Software based binary Instrumentation:

Collect all coverages and share it over share memory with the fuzzer.
fork to have a clean state, fd, memory,…



# KEXT/XNU Fuzzing

Benchmark



# KEXT/XNU Fuzzing

Despite having no tangible runtime overhead, with a little effort, we can embed _sanitizer_cov_trace_pc into trampoline, as well as optimizing _push_regs and _pop_regs away by saving and restoring only **clobbered registers**, we can save some unnecessary CPU cycles per instrumented BB.

Optimizing trampoline

```
void instrument_thunks()
{
    asm volatile (
            ".rept " xstr(REPEAT_COUNT_THUNK) "\n"  // Repeat the following block many times
            "    STR x30, [sp, #-16]!\n"            // save LR. we can't restore it in pop_regs. as we have jumped here.
            "    bl _push_regs\n"
            "    mov x0, #0x0000\n"                 // placeholder targeted_kext flag.
            "    mov x1, #0x4141\n"                 // fix the correct numner when instrumenting as arg0.
            "    mov x1, #0x4141\n"                 // placeholder for BB address
            "    mov x1, #0x4141\n"
            "    mov x1, #0x4141\n"
            "    bl _sanitizer_cov_trace_pc\n"
            "    bl _pop_regs\n"
            "    LDR x30, [sp], #16\n"              // restore LR
            "    nop\n"                             // placeholder for original inst.
            "    nop\n"                             // placeholder for jump back
            ".endr\n"                               // End of repetition
            );
}
```

# KEXT/XNU Fuzzing

```
switch (command.command_case()) {

case Command::kMachPortAllocate: {

  kern_return_t err;
  mach_port_t name = MACH_PORT_NULL;
  err = mach_port_allocate(
      mach_task_self(), command.machportallocate().portright(), &name);
  if (err == KERN_SUCCESS) {
    setElement(mtx_ports, open_ports, name);
  }

  break;
}

case Command::kMachPortInsertRight: {

  mach_port_t name = getElementAtIndex(
      mtx_ports, open_ports, command.machportinsertright().port());
  mach_port_insert_right(mach_task_self(), name, name,
                         command.machportinsertright().msgright());
  break;
}

case Command::kMachPortAllocateName: {
  kern_return_t err;
  err = mach_port_allocate_name(
      mach_task_self(), command.machportallocatename().portright(),
      command.machportallocatename().portname());
  if (err == KERN_SUCCESS) {
    setElement(mtx_ports, open_ports,
               command.machportallocatename().portname());
  }
  break;
}
```

```
DEFINE_TEXT_PROTO_FUZZER(const Session &session) {

  if (fork() == 0) {

    do_fuzz(session);

  } else {

    wait(NULL);
    pishi_collect_in_parent();
  }
}
```

```
syntax = "proto2";

message Session {
  repeated Command commands1 = 1;
  required bytes data_provider = 4;
}

message Command {
  oneof command {
    MachPortNames machPortNames = 1;                 // API: mach_port_names
    MachPortInsertRight machPortInsertRight = 15; // API: mach_port_insert_right
    MachPortAllocateName machPortAllocateName =
        4;                                          // API: mach_port_allocate_name
    MachPortGetRefs machPortGetRefs = 8;           // API: mach_port_get_refs
    MachPortModRefs machPortModRefs = 9;           // API: mach_port_mod_refs
    MachPortDestroy machPortDestroy = 6;           // API: mach_port_destroy
    MachPortDeallocate machPortDeallocate = 7;  // API: mach_port_deallocate
    MachPortDestruct machPortDestruct = 33;       // API: mach_port_destruct
    MachPortAllocate machPortAllocate = 5;        // API: mach_port_allocate
    MachPortExtractRight machPortExtractRight =
```

# KEXT/XNU Fuzzing

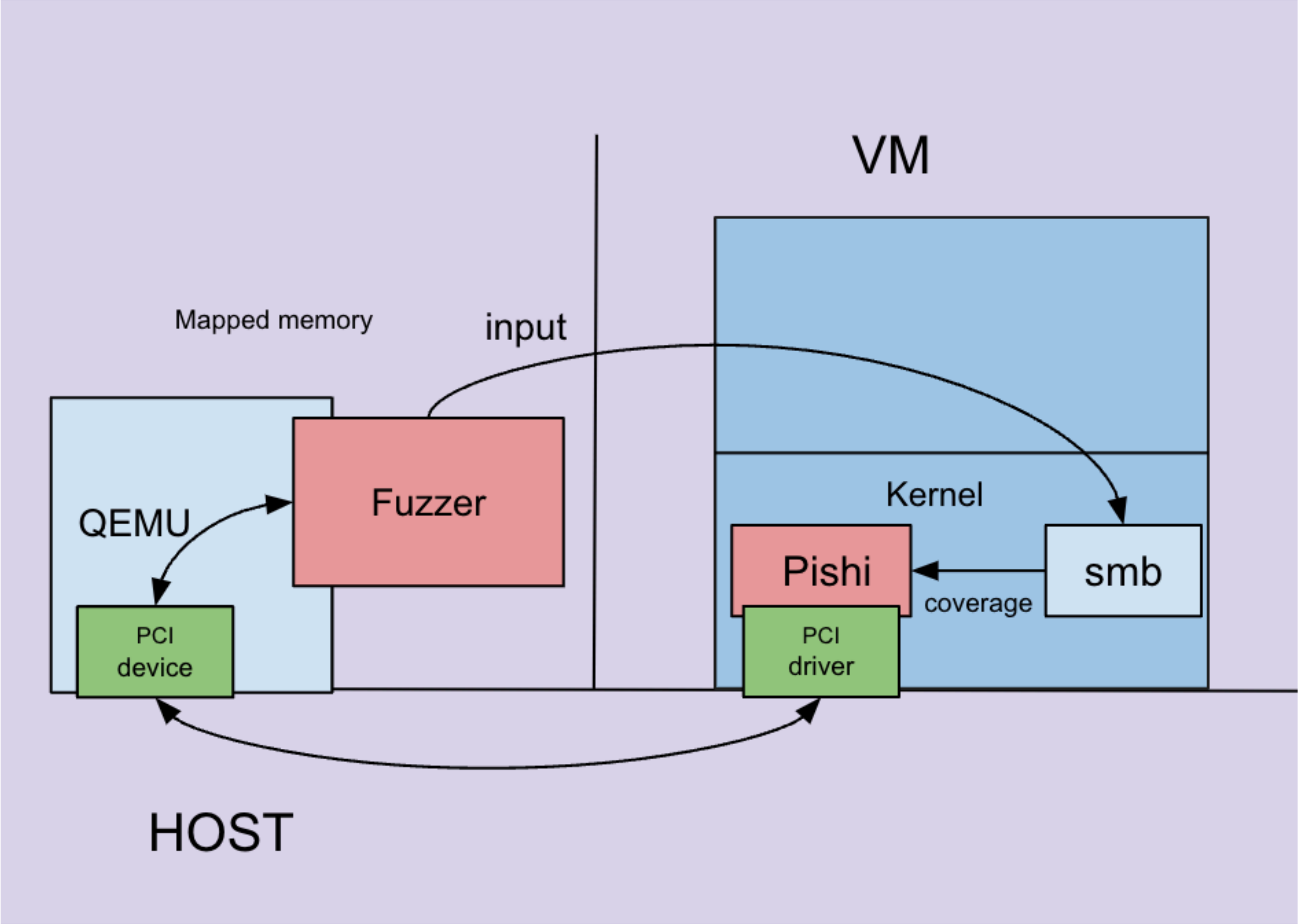
```
panic(cpu 1 caller 0xfffffe0015cbee28): PAC failure from kernel with DA key while authing x16 at pc 0xfffffe001549d268, lr 0x95bcfe001548e020
      x0:  0x0000000000000001 x1:  0x0000000000001a03  x2:  0xfffffe24ccd3c670  x3:  0x0000000000000013
      x4:  0xfffffe24ccd3c670 x5:  0xfffffe401e3afa6c  x6:  0xfffffe401e3afd10  x7:  0xfffffe401e3afc60
      x8:  0x0000000000100000 x9:  0x0000000003100001  x10: 0xfffffe1b33ae8400  x11: 0x0000000000000288
      x12: 0x0000000000000011 x13: 0x0000000000000000  x14: 0x0000000000000000  x15: 0x0000000000000000
      x16: 0x0000000000000000 x17: 0xf444fe24ccd3c670  x18: 0x0000000000000000  x19: 0x0000000000001a03
      x20: 0xfffffe401e3afa6c x21: 0xfffffe1b3480db50  x22: 0xfffffe401e3afc68  x23: 0x0000000000000013
      x24: 0xfffffe1b3480db50 x25: 0x0000000000000013  x26: 0xfffffe1b3480db50  x27: 0x0000000000131313
      x28: 0x0000000010000003 fp:  0xfffffe401e3af9d0  lr:  0x95bcfe001548e020  sp:  0xfffffe401e3af9a0
      pc:  0xfffffe001549d268 cpsr: 0x20401204         esr: 0x72000002          far: 0x0000000102600000

Debugger message: panic
Memory ID: 0x0
OS release type: User
OS version: 23F79
Kernel version: Darwin Kernel Version 23.5.0: Wed May  1 20:12:39 PDT 2024; root:xnu-10063.121.3~5/RELEASE_ARM64_VMAPPLE
Fileset Kernelcache UUID: 29397EDDD6C60A125AA3CC4EC8D6148A
Kernel UUID: A8517A76-B187-30FE-ADF3-0303CDEE33CE
Boot session UUID: 430D3164-42FA-416B-9AC4-B57644CFB5A3
```

# KEXT/XNU Fuzzing

virtIO-shmem

ivshmem

Fuzzing remote attack surfaces like SMB



# KEXT/XNU Fuzzing

Thank you for listening!

I have covered a lot more in my blog post( going to publish it just now)

R00tkitsmm.github.io

Any questions?