Nd Neodyme

Revisiting Widevine L3

DRM as a Playground for Hackers

Hack.lu 2025

Felipe Custodio
@_localo_

Nd Neodyme

01

DRM 101

DRM 101 - Overview

Apple	Google	Microsoft
		
FairPlay	Widevine	PlayReady

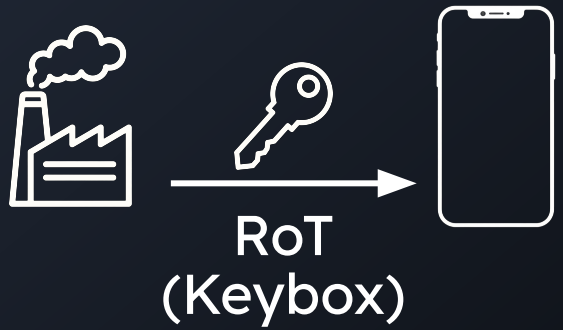
DRM 101 - Overview

Apple	Google	Microsoft
		
FairPlay	Widevine	PlayReady

DRM 101 - Widevine (Simplified)

Device Provisioning

DRM 101 - Widevine (Simplified)

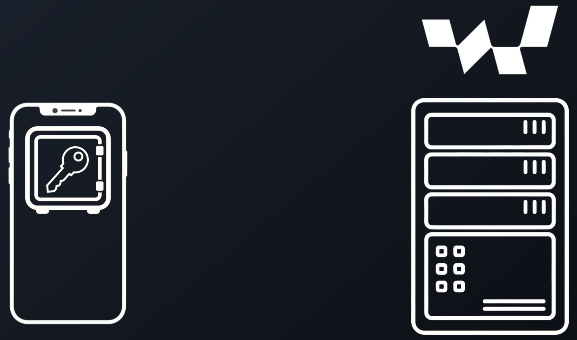


Device Provisioning

DRM 101 - Widevine (Simplified)

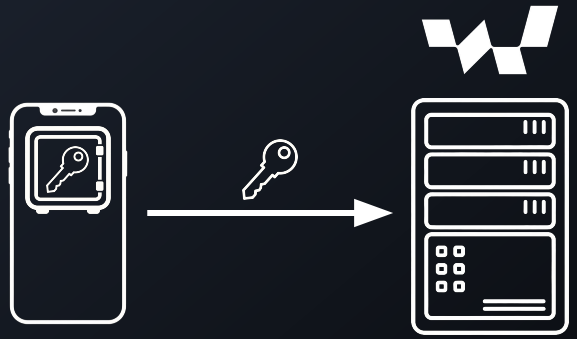
Device Provisioning

DRM 101 - Widevine (Simplified)



Certificate Provisioning

DRM 101 - Widevine (Simplified)



Certificate Provisioning

DRM 101 - Widevine (Simplified)

Certificate Provisioning

DRM 101 - Widevine (Simplified)

Certificate Provisioning

DRM 101 - Widevine (Simplified)

Content Provisioning

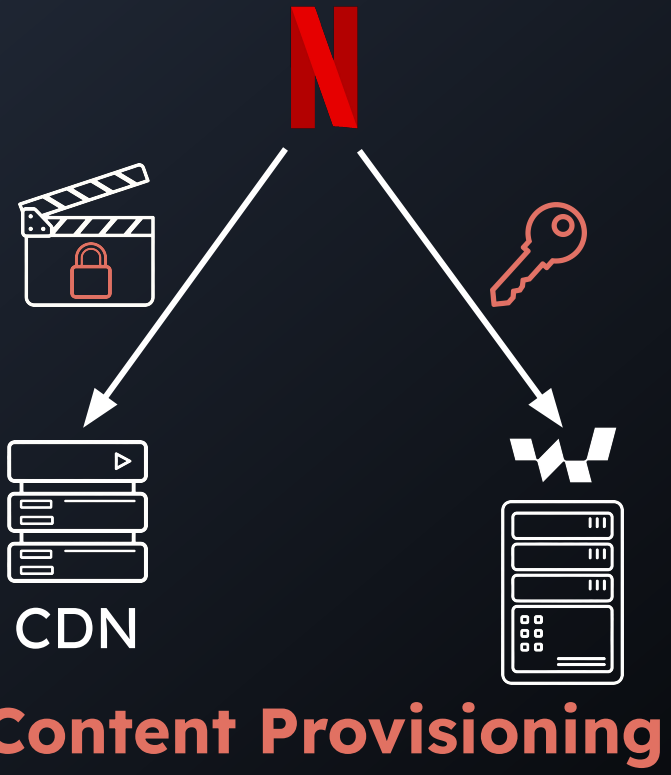
DRM 101 - Widevine (Simplified)

Content Provisioning

DRM 101 - Widevine (Simplified)

Content Provisioning

DRM 101 - Widevine (Simplified)

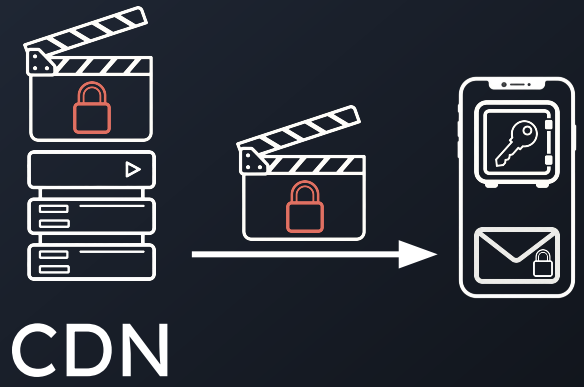


DRM 101 - Widevine (Simplified)

CDN

License Provisioning

DRM 101 - Widevine (Simplified)



License Provisioning

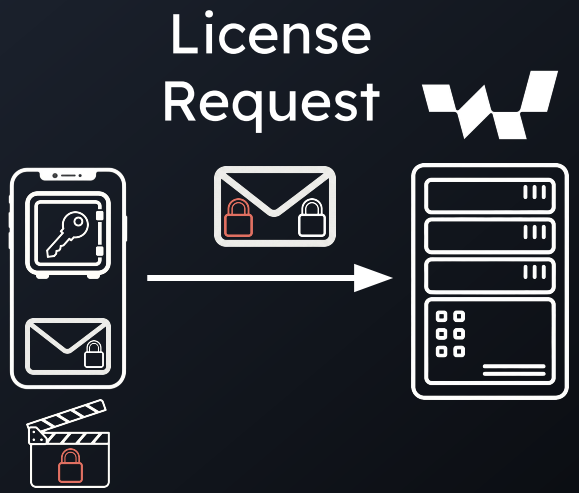
DRM 101 - Widevine (Simplified)

License Provisioning

DRM 101 - Widevine (Simplified)

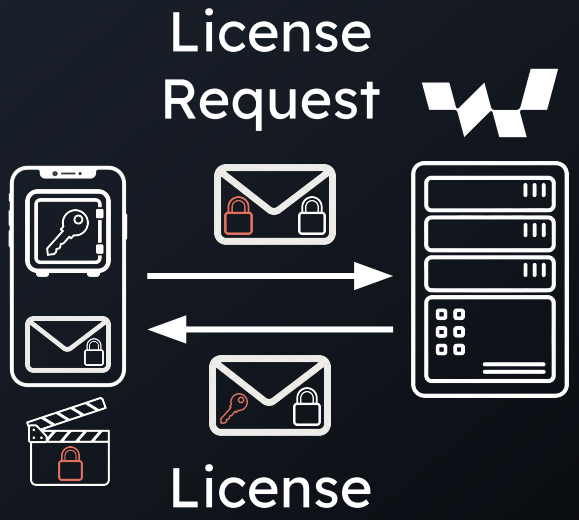
License Provisioning

DRM 101 - Widevine (Simplified)



License Provisioning

DRM 101 - Widevine (Simplified)



License Provisioning

DRM 101 - Widevine (Simplified)

License Provisioning

DRM 101 - Widevine (Simplified)

Content Decryption

DRM 101 - Widevine (Simplified)

Content Decryption

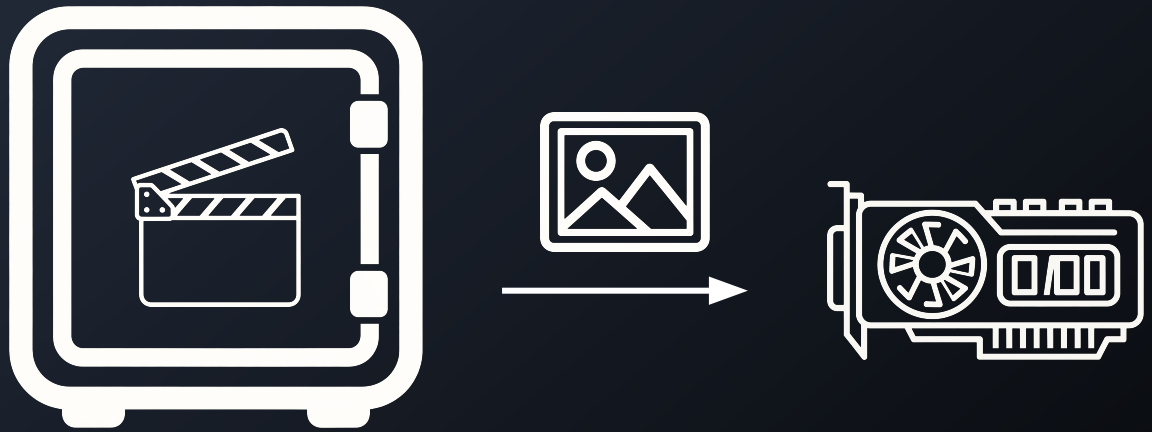
DRM 101 - Widevine (Simplified)

Content Decryption

DRM 101 - Widevine (Simplified)

Content Decryption

DRM 101 - Widevine (Simplified)



Netflix and Chill

DRM 101 - Widevine

	L1	L2	L3
Decryption	TrustZone	TrustZone	Software
Processing	TrustZone	Software	Software

DRM 101 - Widevine Level 3

Interface: EME
L3: libwidevinecdm.so

Interface: OEM Crypto
L3: libwvhidl.so*

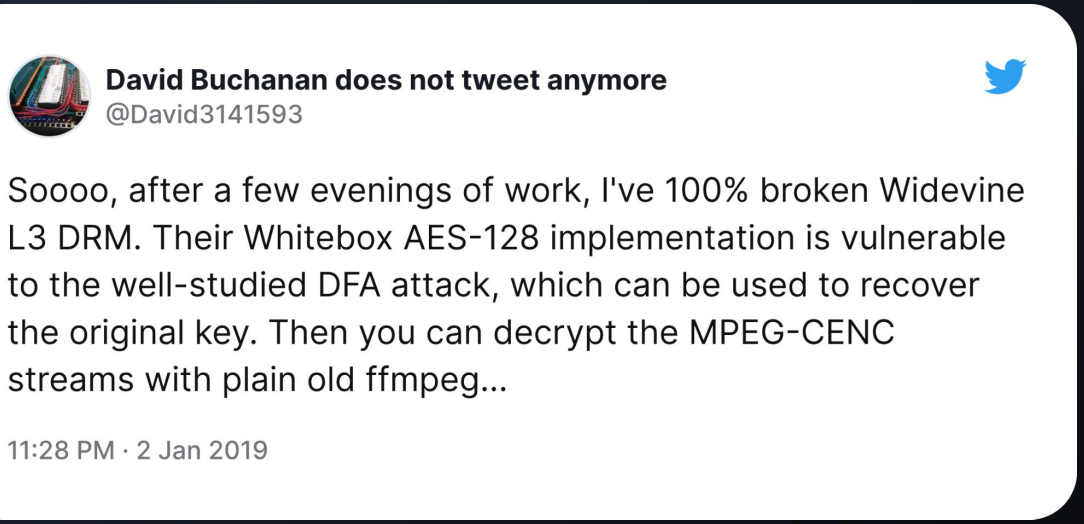
* Depends on Android version

Nd Neodyme

02

Prior Work

Prior Work - Break L3 using DFA



Seems like I'm late to the party..

Prior Work - Break L3 using Reverse Engineering

Reversing the old Widevine Content Decryption Module

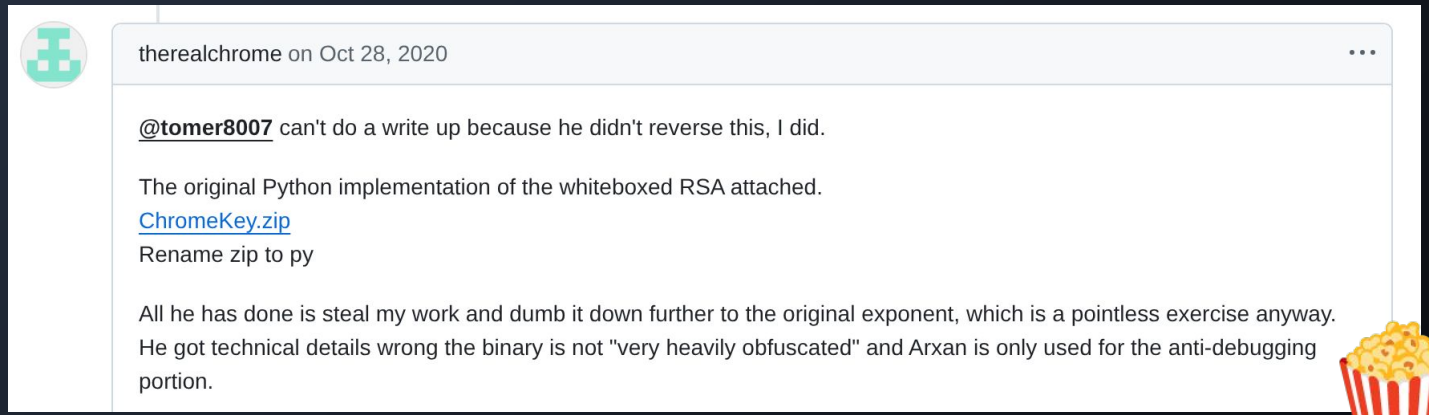
Tomer edited this page on Jul 28, 2021 · 6 revisions

This write-up describes the process that was done reversing Widevine's *old* Windows CDM (`widevinecdm.dll` 4.10.1610.0) to bypass its protection and extract its RSA private key. Knowing the private key could eventually lead to the decryption of media content keys for L3.

Everything is for educational purposes only, and the techniques described here will **NOT** work against newer versions of the CDM anyway since it had gone through a major refactoring, i.e obfuscation techniques & algorithms were changed. If all you care about is ripping videos, please look elsewhere.

"someone" reverse-engineered a newer version of L3

Prior Work - Break L3 using Reverse Engineering



therealchrome on Oct 28, 2020

@tomer8007 can't do a write up because he didn't reverse this, I did.

The original Python implementation of the whiteboxed RSA attached.
ChromeKey.zip
Rename zip to py

All he has done is steal my work and dumb it down further to the original exponent, which is a pointless exercise anyway. He got technical details wrong the binary is not "very heavily obfuscated" and Arxan is only used for the anti-debugging portion.

"someone" reverse-engineered a newer version of L3

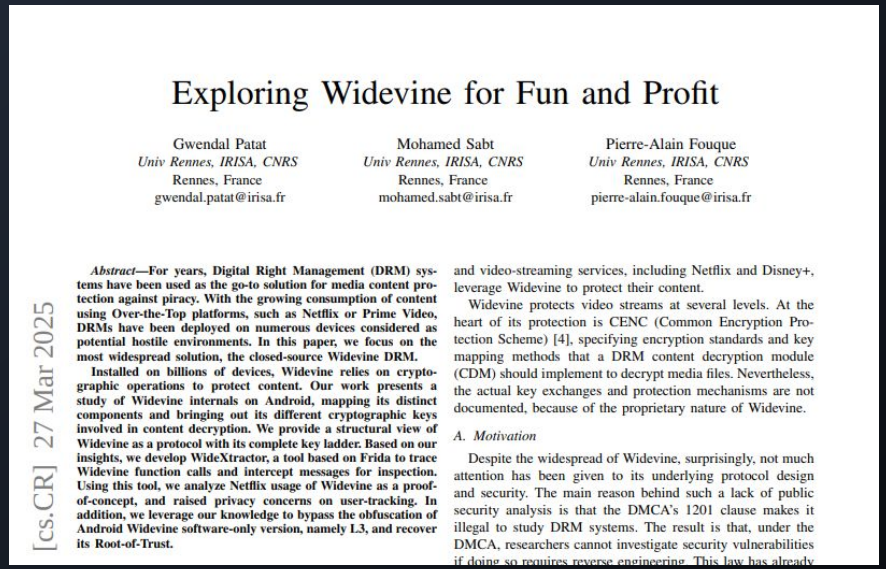
Prior Work - Pirating Tutorial

Good starting point!

Describes a working setup to dump the key

Prior Work - Scientific Research

Really nice for a more detailed explanation of the protocol



Exploring Widevine for Fun and Profit

Gwendal Patat
Univ Rennes, IRISA, CNRS
Rennes, France
gwendal.patat@irisa.fr

Mohamed Sabt
Univ Rennes, IRISA, CNRS
Rennes, France
mohamed.sabt@irisa.fr

Pierre-Alain Fouque
Univ Rennes, IRISA, CNRS
Rennes, France
pierre-alain.fouque@irisa.fr

***Abstract*—For years, Digital Right Management (DRM) systems have been used as the go-to solution for media content protection against piracy. With the growing consumption of content using Over-the-Top platforms, such as Netflix or Prime Video, DRMs have been deployed on numerous devices considered as potential hostile environments. In this paper, we focus on the most widespread solution, the closed-source Widevine DRM.**

Installed on billions of devices, Widevine relies on cryptographic operations to protect content. Our work presents a study of Widevine internals on Android, mapping its distinct components and bringing out its different cryptographic keys involved in content decryption. We provide a structural view of Widevine as a protocol with its complete key ladder. Based on our insights, we develop WideXtractor, a tool based on Frida to trace Widevine function calls and intercept messages for inspection. Using this tool, we analyze Netflix usage of Widevine as a proof-of-concept, and raised privacy concerns on user-tracking. In addition, we leverage our knowledge to bypass the obfuscation of Android Widevine software-only version, namely L3, and recover its Root-of-Trust.

and video-streaming services, including Netflix and Disney+, leverage Widevine to protect their content.

Widevine protects video streams at several levels. At the heart of its protection is CENC (Common Encryption Protection Scheme) [4], specifying encryption standards and key mapping methods that a DRM content decryption module (CDM) should implement to decrypt media files. Nevertheless, the actual key exchanges and protection mechanisms are not documented, because of the proprietary nature of Widevine.

A. Motivation

Despite the widespread of Widevine, surprisingly, not much attention has been given to its underlying protocol design and security. The main reason behind such a lack of public security analysis is that the DMCA's 1201 clause makes it illegal to study DRM systems. The result is that, under the DMCA, researchers cannot investigate security vulnerabilities if doing so requires reverse engineering. This law has already

The Widevine Playground

A

Key extraction using Differential Fault Analysis

B

Reverse engineering the obfuscation

C

Dumping the keybox from memory

The Widevine Playground

A

Key extraction using Differential Fault Analysis

B

Reverse engineering the obfuscation

C

Dumping the keybox from memory

Nd Neodyme

03

Dumping the Keybox from Memory

Dumping the Keybox from Memory

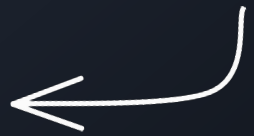
Using Frida

1. Install Android emulator
2. Run Frida script which hooks munmap
3. Scan memory for keybox
4. Profit?

Dumping the Keybox from Memory

Using Frida

There must be a more lightweight approach



1. Install Android emulator
2. Run Frida script which hooks munmap
3. Scan memory for keybox
4. Profit?

Dumping the Keybox from Memory

Using Qiling

Qiling is a binary emulation framework
The OS layer is replaced with python code

Supports many operating systems:
Linux, Windows, Android, ...

Dumping the Keybox from Memory

Using Qiling

LIEF can trick the linker into thinking it’s an executable

```
lib = lief.parse(src)

lib[lief.ELF.DynamicEntry.TAG.from_value(0x6000000F)].value = (
    lib[lief.ELF.DynamicEntry.TAG.from_value(0x6000000F)].value + 0x1000
)
lib.interpreter = b"/system/bin/linker"

lib.write(dst)
```

Dumping the Keybox from Memory

Using Qiling

Set up the emulator context for Android

```
env = {"ANDROID_DATA": r"/data", "ANDROID_ROOT": r"/system"}
OVERRIDES = {"mmap_address": 0x68000000}
ql = Qiling(
    ["./rootfs/libwvhidl.so.exe"],
    "rootfs",
    env,
    ostype=QL_OS.LINUX,
    verbose=QL_VERBOSE.OFF,
    profile={"OS32": OVERRIDES},
    multithread=True,
)
```

Dumping the Keybox from Memory

Using Qiling

Stub missing syscalls

```python
def hook_fstatat64(ql: Qiling, dirfd: int, path: int, buf_ptr: int, flags int):
    return 0

def hook_fstatfs64(ql: Qiling, dirfd: int, path: int, buf_ptr: int, flags int):
    return 0

ql.os.set_syscall("fstatat64", hook_fstatat64, QL_INTERCEPT.CALL)
ql.os.set_syscall("fstatfs64", hook_fstatfs64, QL_INTERCEPT.CALL)
ql.os.set_syscall("mkdirat", ql_syscall_mkdirat, QL_INTERCEPT.CALL)
```

Dumping the Keybox from Memory

Using Qiling

Replace complex Android functions with our own

```python
def hook_log_write(ql: Qiling):
    params = ql.os.resolve_fcall_params(
       { "priority": PARAM_INT32, "tag": STRING, "message": STRING}
    )
    print(
        f"LOG_{LOG_PRIORITY[params['priority']]:8s} [{params['tag']}] {params['message']}"
    )

    ql.arch.regs.arch_pc = ql.arch.stack_pop()

ql.os.set_api("__android_log_write", hook_log_write, QL_INTERCEPT.CALL)
```

Dumping the Keybox from Memory

Using Qiling

Let the loader do its magic and execute our target function

```
ql.run(end=0x56555000)  # stop at main

ql.stack_push(0x41424344)
ql.emu_start(
    begin=get_function(ql, L3_SYMBOL_MAPPING["Initialize"]), end=0x41424344
)
print(f"L3 Initialize done: {OEM_CPRYPTO_RESULTS[ql.arch.regs.eax]}" )
```

Dumping the Keybox from Memory

Using Qiling

Let the loader do its magic and execute our target function

This is L3 log output!

```
linker: warning: unable to get realpath for the library "/system/lib/libvndksupport.so".Will use given path
[!] [Thread 2000]   Nothing to do but still get scheduled!
LOG_INFO     [WVCdm] [(0):] Level3 Library 4464 Apr 20 2018 14:54:35
LOG_WARN     [WVCdm] [file_store.cpp(149):Open] File::Open: fopen failed: 1
LOG_WARN     [WVCdm] [file_store.cpp(149):Open] File::Open: fopen failed: 1
L3 Initialize done: SUCCESS
```

Dumping the Keybox from Memory

Keybox Structure

	Size (Bytes)
Device ID	0x20
Device Key	0x10
Data (version specific)	0x48
Magic (“kbox”)	0x4
Checksum	0x4
	Total: 0x80

Dumping the Keybox from Memory

Using Qiling

And now it’s just simple memory scan ;)

```python
def hook_munmap(ql: Qiling):
    candidates = ql.mem.search(b"\x6b\x62\x6f\x78")
    for candidate in candidates:
        keybox_addr = candidate - (72 + 16 + 32)
        keybox = ql.mem.read(keybox_addr, 128)
        device_id, device_key, data, magic, crc = struct.unpack("<32s16s72sII", keybox)
        assert magic == 0x786F626B
        print("Keybox found at:", hex(keybox_addr))
        print("Device ID:", device_id.strip().decode())
        print("kbox: ", keybox.hex())
        ql.arch.regs.eax = 0
        ql.emu_stop()

ql.os.set_api("munmap", hook_munmap, QL_INTERCEPT.ENTER)
```

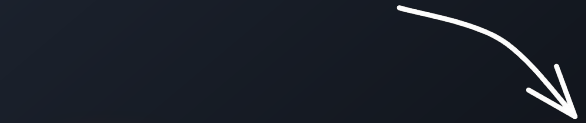
Dumping the Keybox from Memory

Recap

1. Convert the libwvhidl.so library to an executable
2. Use Qiling to load the executable
3. Hook munmap
4. Run the Widevine L3 initialization function
5. The memory is not cleared before munmap
6. Leak the keybox by scanning for the “kbox” magic bytes

The Widevine Playground

That was easier than expected



A

Key extraction using Differential Fault Analysis

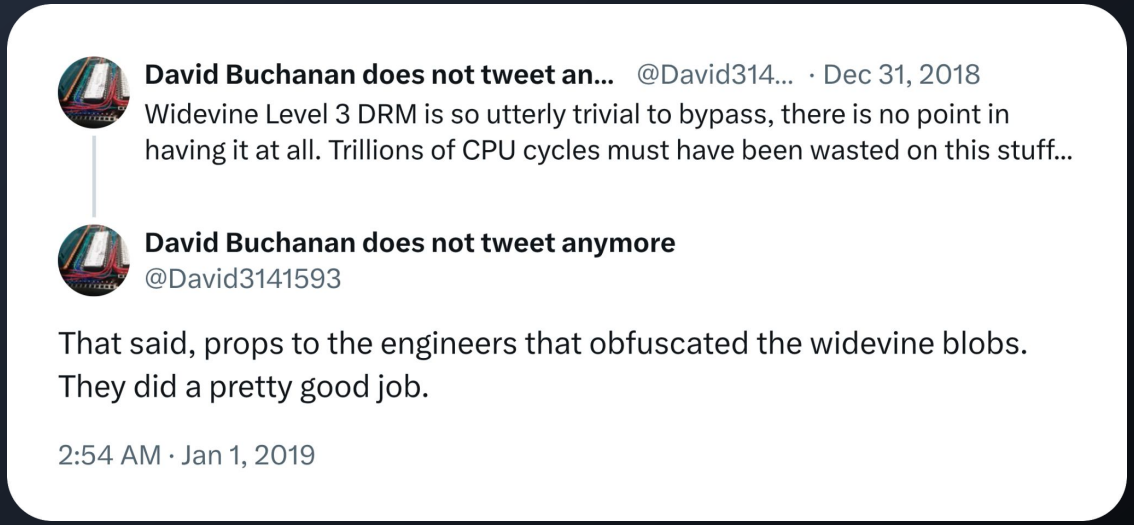
B

Reverse engineering the obfuscation

C

Dumping the keybox from memory

Strong obfuscation?



David Buchanan does not tweet an... @David314... · Dec 31, 2018
Widevine Level 3 DRM is so utterly trivial to bypass, there is no point in having it at all. Trillions of CPU cycles must have been wasted on this stuff...

David Buchanan does not tweet anymore
@David3141593

That said, props to the engineers that obfuscated the widevine blobs. They did a pretty good job.

2:54 AM · Jan 1, 2019

The Widevine Playground

A

Key extraction using Differential Fault Analysis

B

Reverse engineering the obfuscation

C

Dumping the keybox from memory

Nd Neodyme

04

AES DFA

What to attack?

Each vendor gets a unique version of the L3 code

The keybox is generated on first use

The keybox is stored encrypted as ay64.dat using a unique key

What to attack?

Each vendor gets a unique version of the L3 code

The keybox is generated on first use

The keybox is stored encrypted as ay64.dat using a unique key

We want this key!

AES DFA in 5 simple Steps

1. Identify the AES operations
2. Inject a fault at a precise location that corrupts the internal state
3. Generate many faulted outputs for the same input
4. Throw the result into phoenixAES
5. Profit?

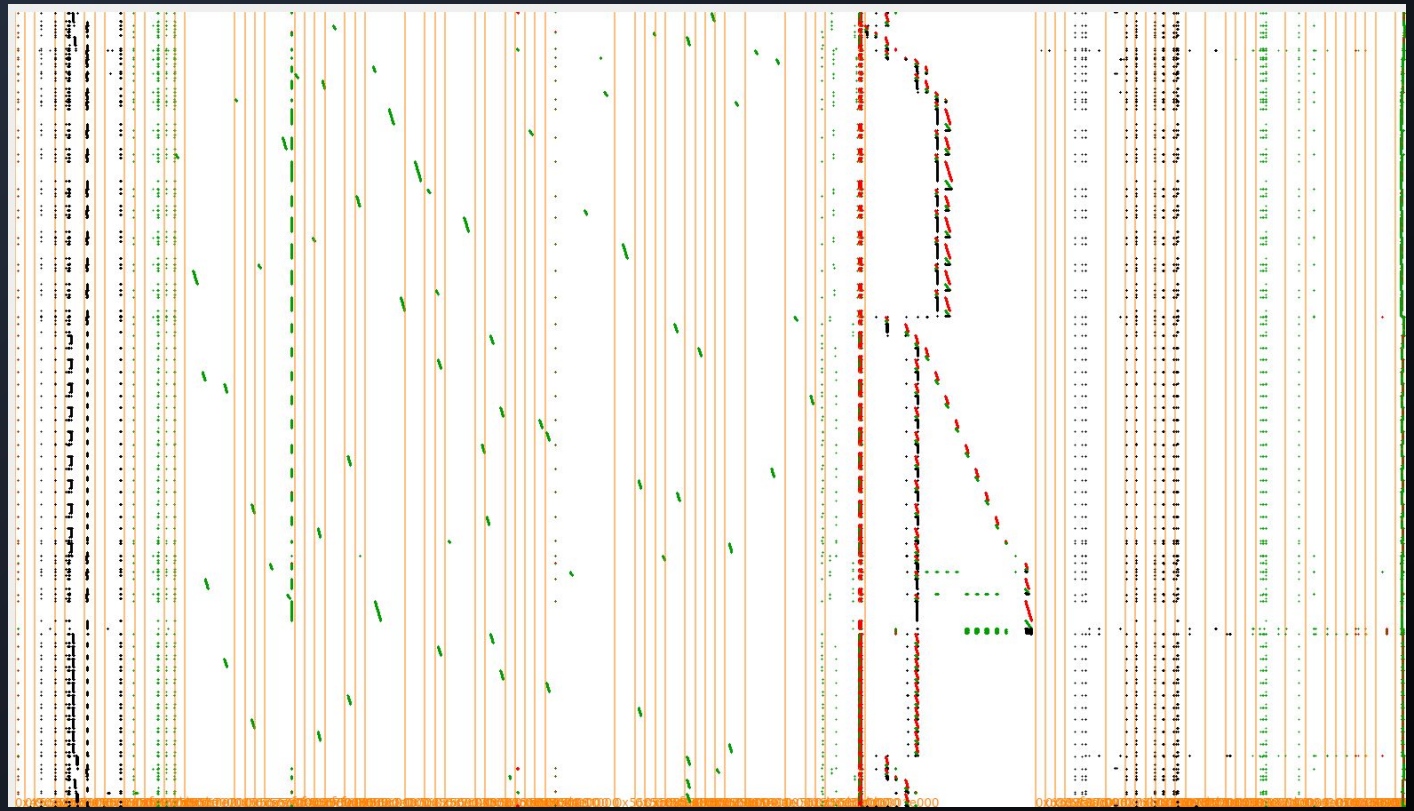
AES DFA in 5 simple Steps

TraceGraph can be used to display memory reads and writes on an execution trace.

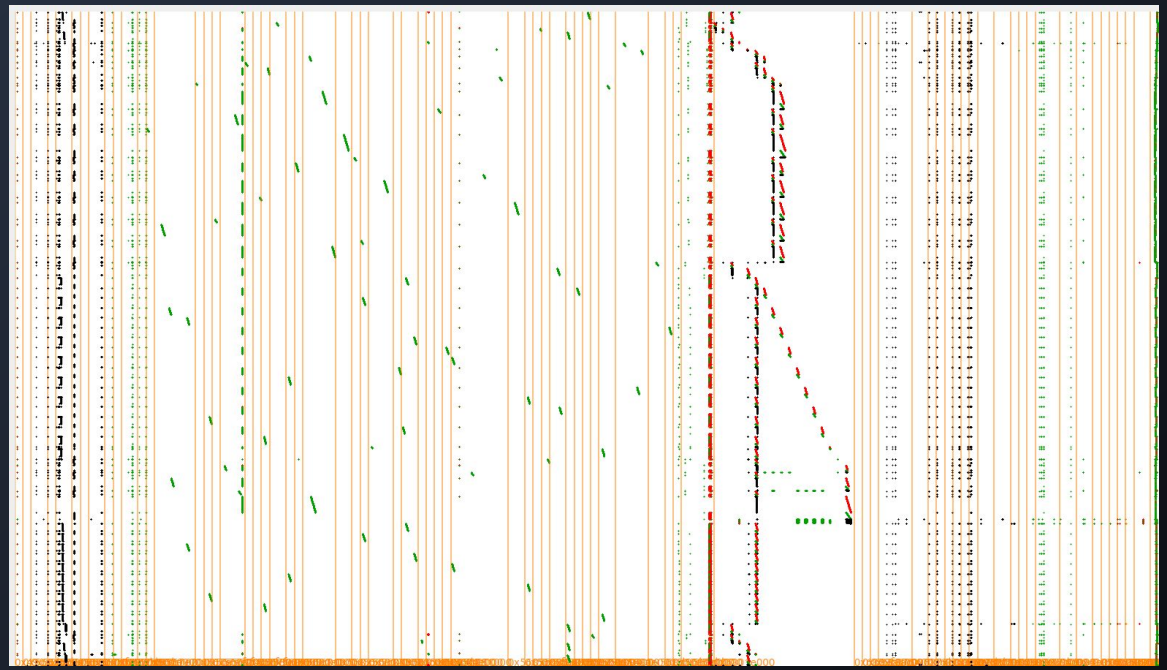
The trace is loaded from an SQLite database.

We can use Qiling to create such traces!

Finding the Fault Position - TraceGraph

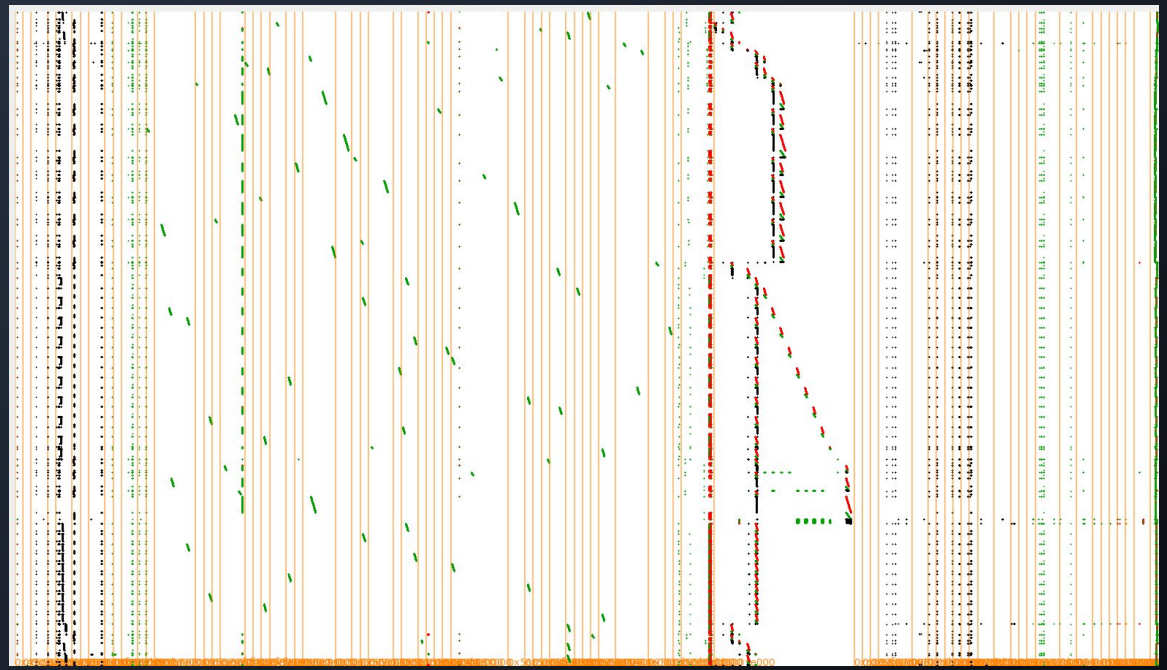


Finding the Fault Position - TraceGraph



Time

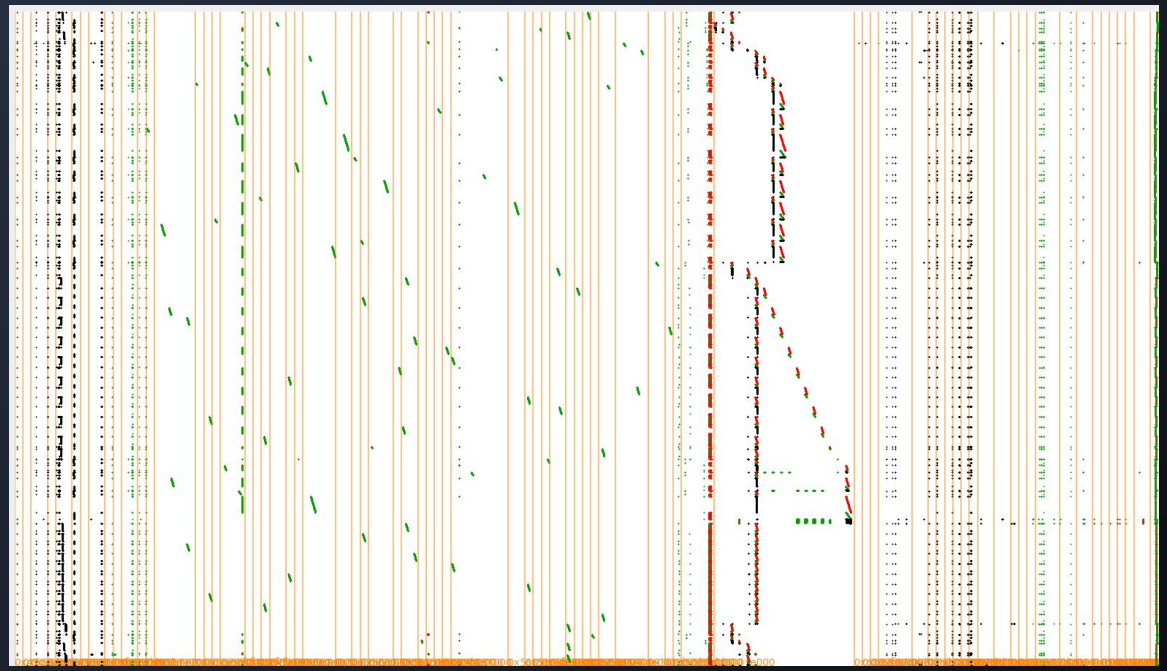
Finding the Fault Position - TraceGraph



Time

Address

Finding the Fault Position - TraceGraph



Time

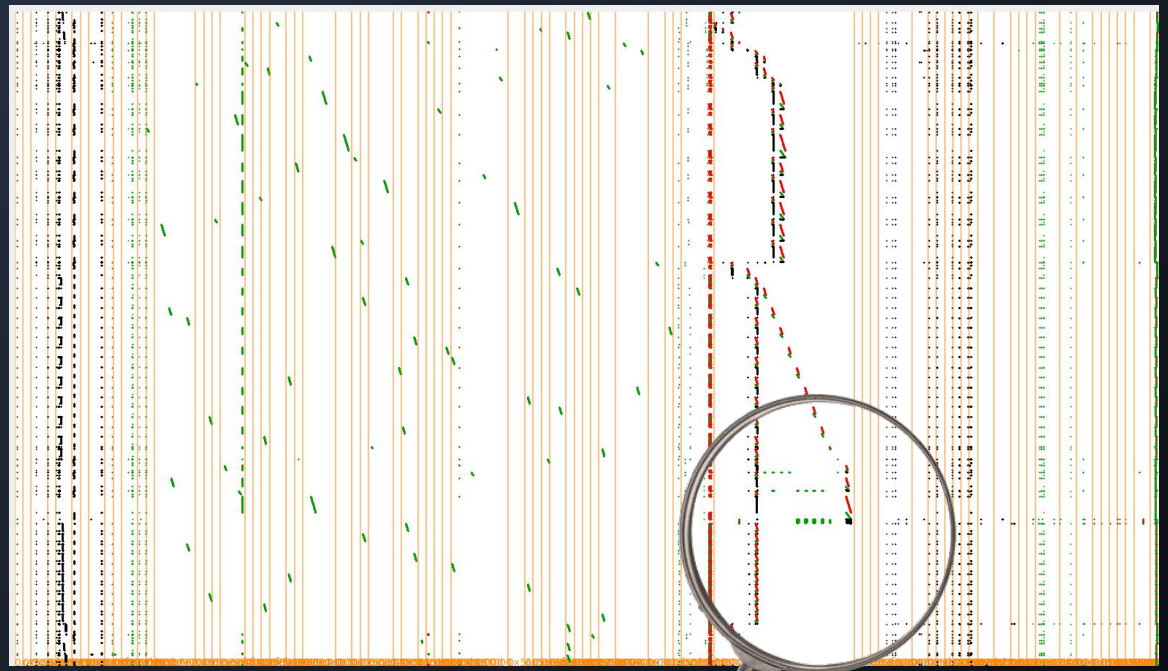
Write

Read

Execute

Address

Finding the Fault Position - TraceGraph



Time

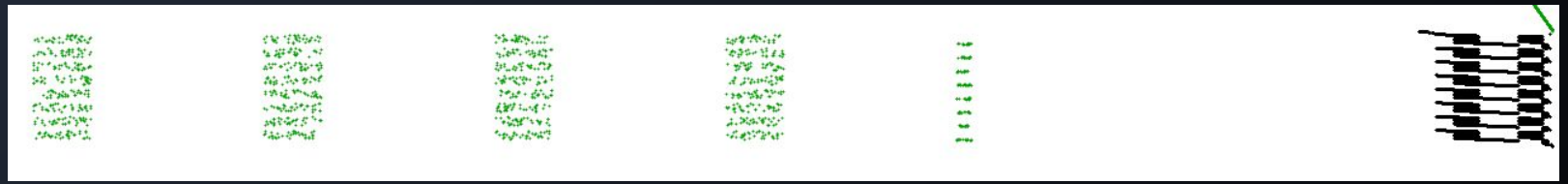
Write

Read

Execute

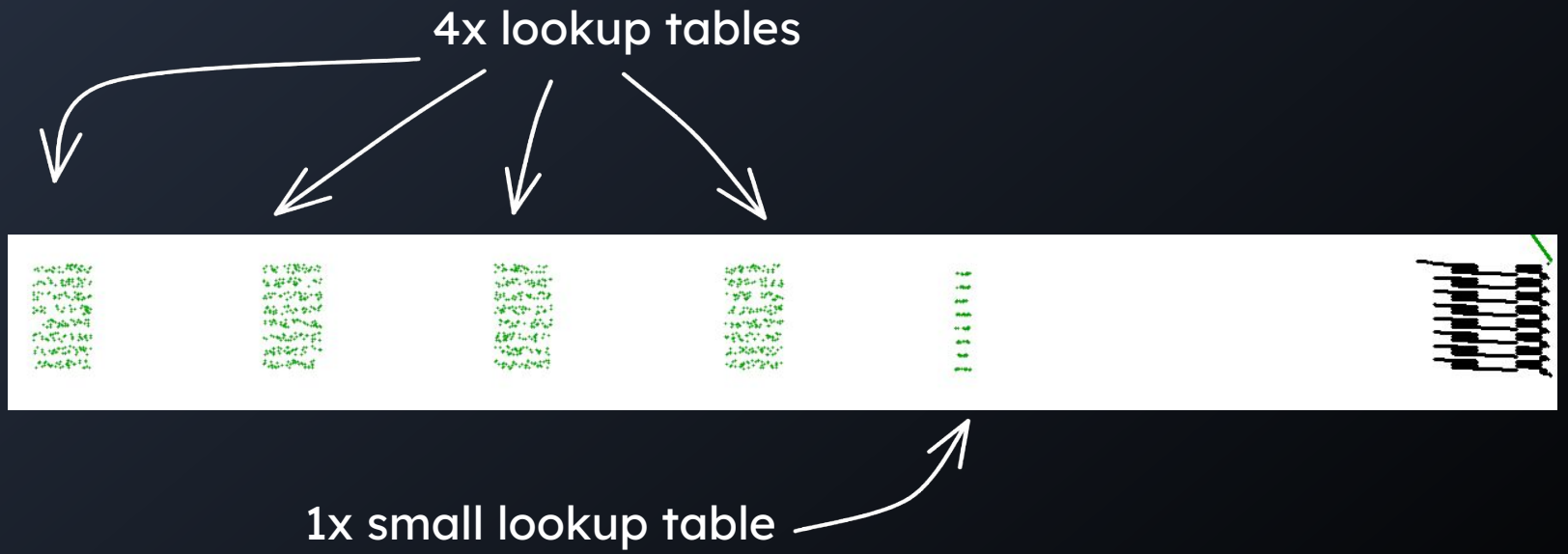
Add

Finding the Fault Position - AES

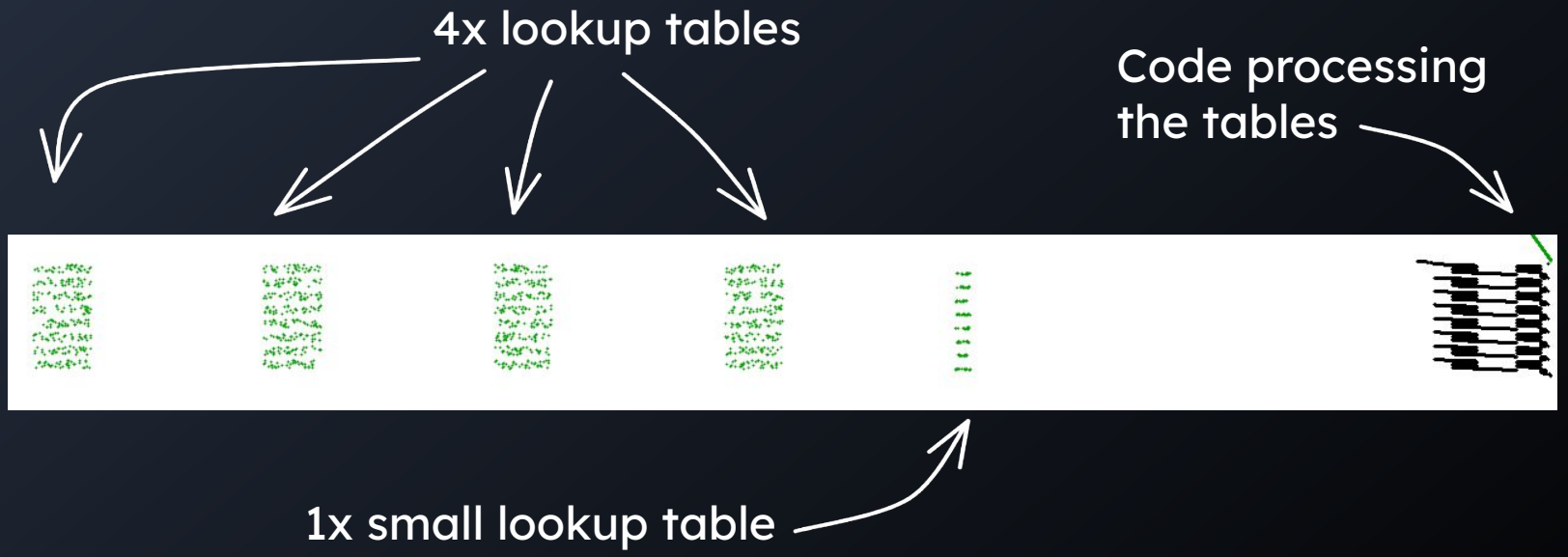


Finding the Fault Position - AES

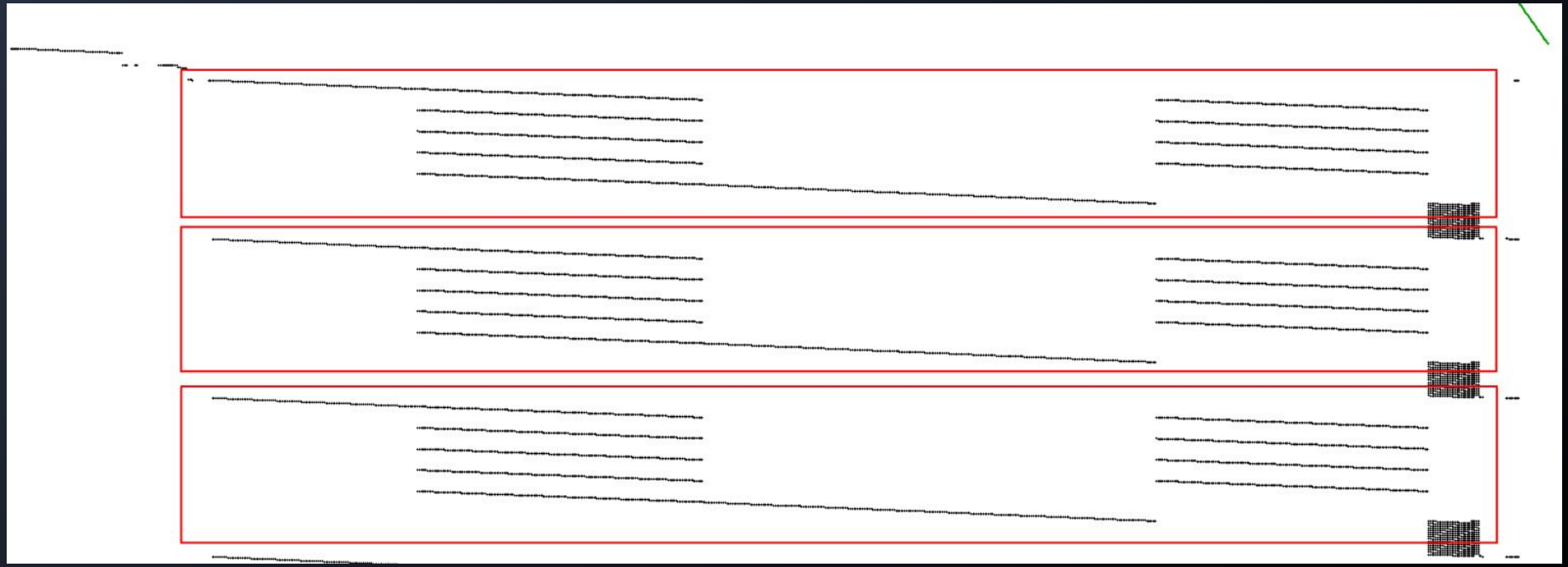
Finding the Fault Position - AES



Finding the Fault Position - AES



Finding the Fault Position - AES



Some large loop

Finding the Fault Position - AES

```
KeyExpansion
AddRoundKey
for i in range rounds-1:
    Substitution
    ShiftRows
    MixColumns
    AddRoundKey
Substitution
ShiftRows
AddRoundKey
```

Finding the Fault Position - AES

KeyExpansion

AddRoundKey

for i in range rounds-1:

    Substitution

    ShiftRows

    MixColumns

    AddRoundKey

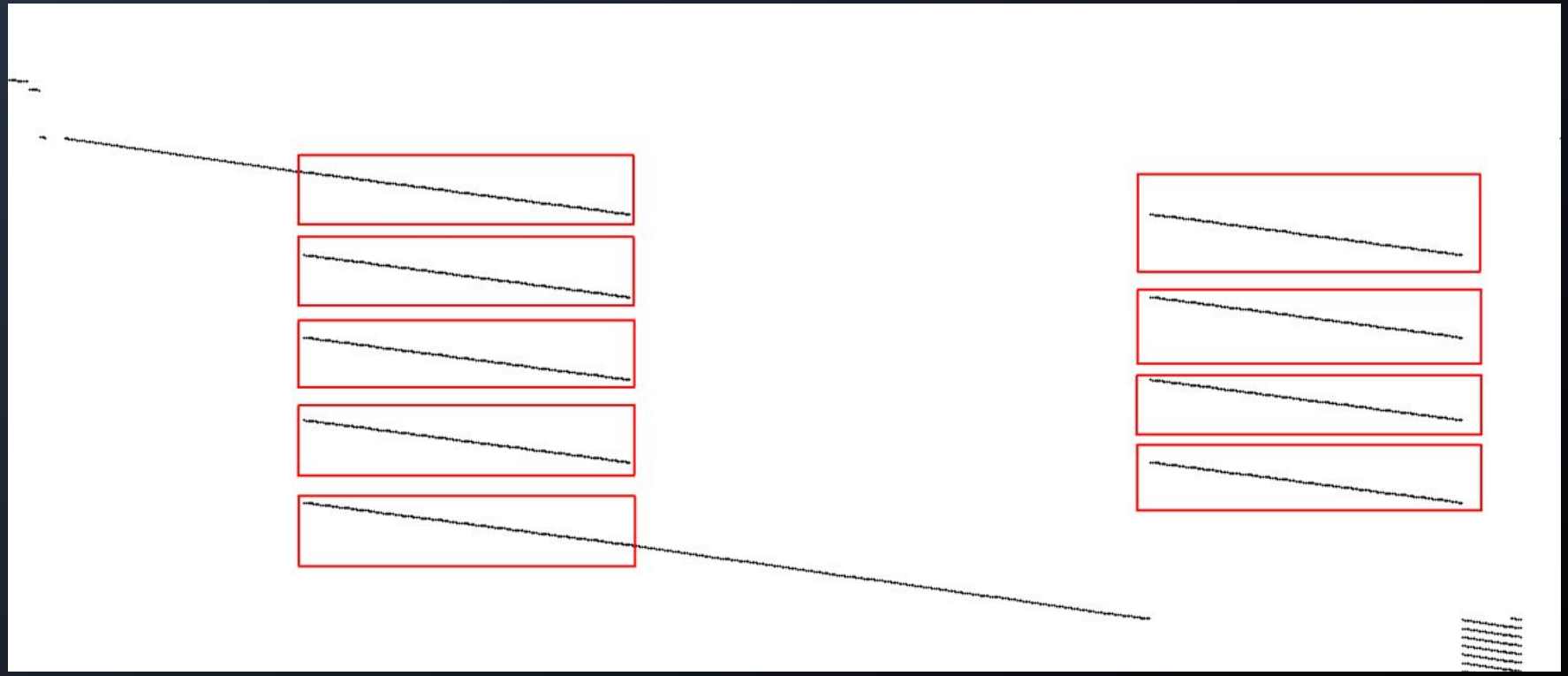
Substitution

ShiftRows

AddRoundKey

This inner loop should be easy to spot, same addresses executed over and over again

Finding the Fault Position - AES



Two blocks repeated, 5+4=9 times

Finding the Fault Position - AES

```
KeyExpansion
AddRoundKey
for i in range rounds-1:
    Substitution
    ShiftRows
    MixColumns
    AddRoundKey
Substitution
ShiftRows
AddRoundKey
```

Finding the Fault Position - AES

```
KeyExpansion
AddRoundKey
for i in range rounds-1:
    Substitution
    ShiftRows
    MixColumns
    AddRoundKey
Substitution
ShiftRows
AddRoundKey
```

9 repetitions

10 rounds

Finding the Fault Position - AES

```
KeyExpansion
AddRoundKey
for i in range rounds-1:
    Substitution
    ShiftRows
    MixColumns
    AddRoundKey
Substitution
ShiftRows
AddRoundKey
```

9 repetitions

10 rounds

AES-128

Finding the Fault Position - AES

KeyExpansion

AddRoundKey

for i in range rounds-1:

    Substitution

    ShiftRows

    MixColumns

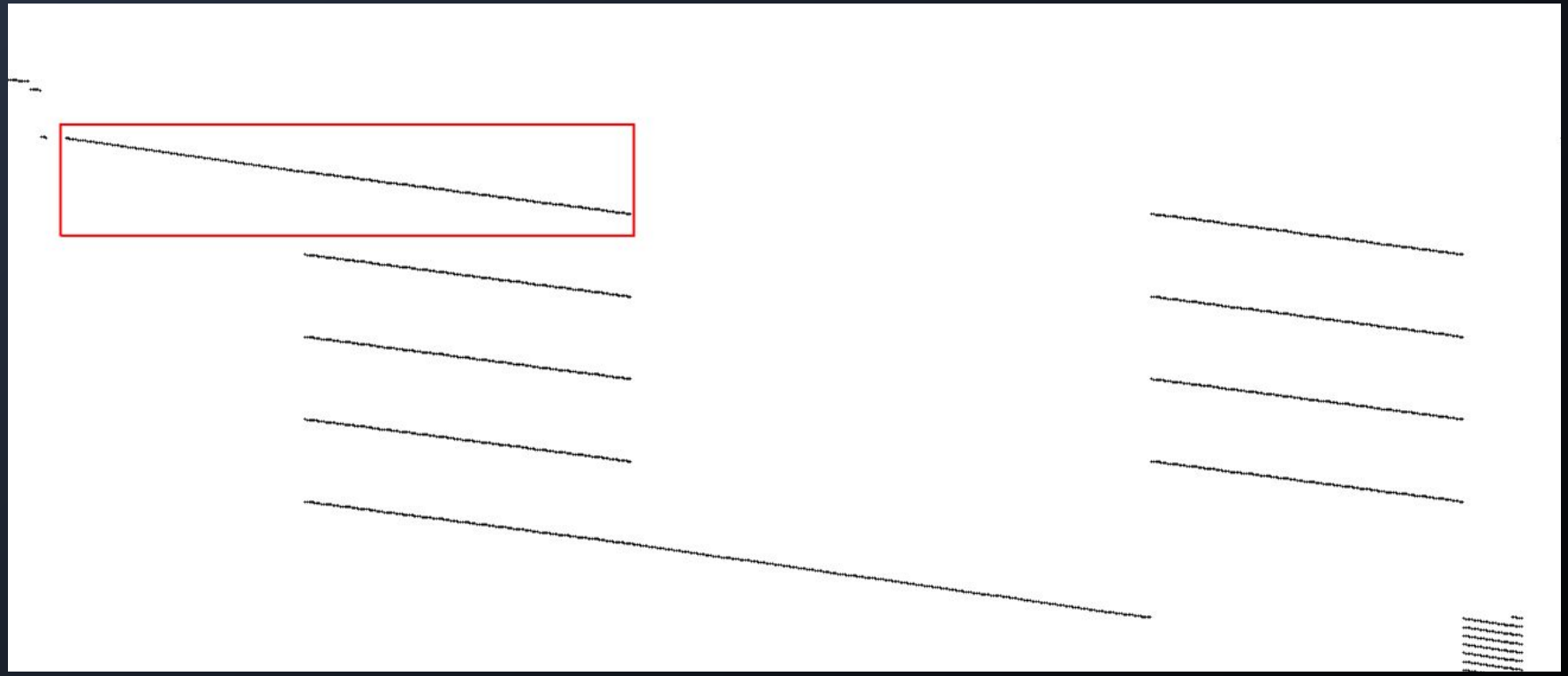
    AddRoundKey

Substitution

ShiftRows

AddRoundKey

Finding the Fault Position - AES



AES setup and first round

Finding the Fault Position - AES

KeyExpansion

AddRoundKey

for i in range rounds-1:

    Substitution

    ShiftRows

    MixColumns

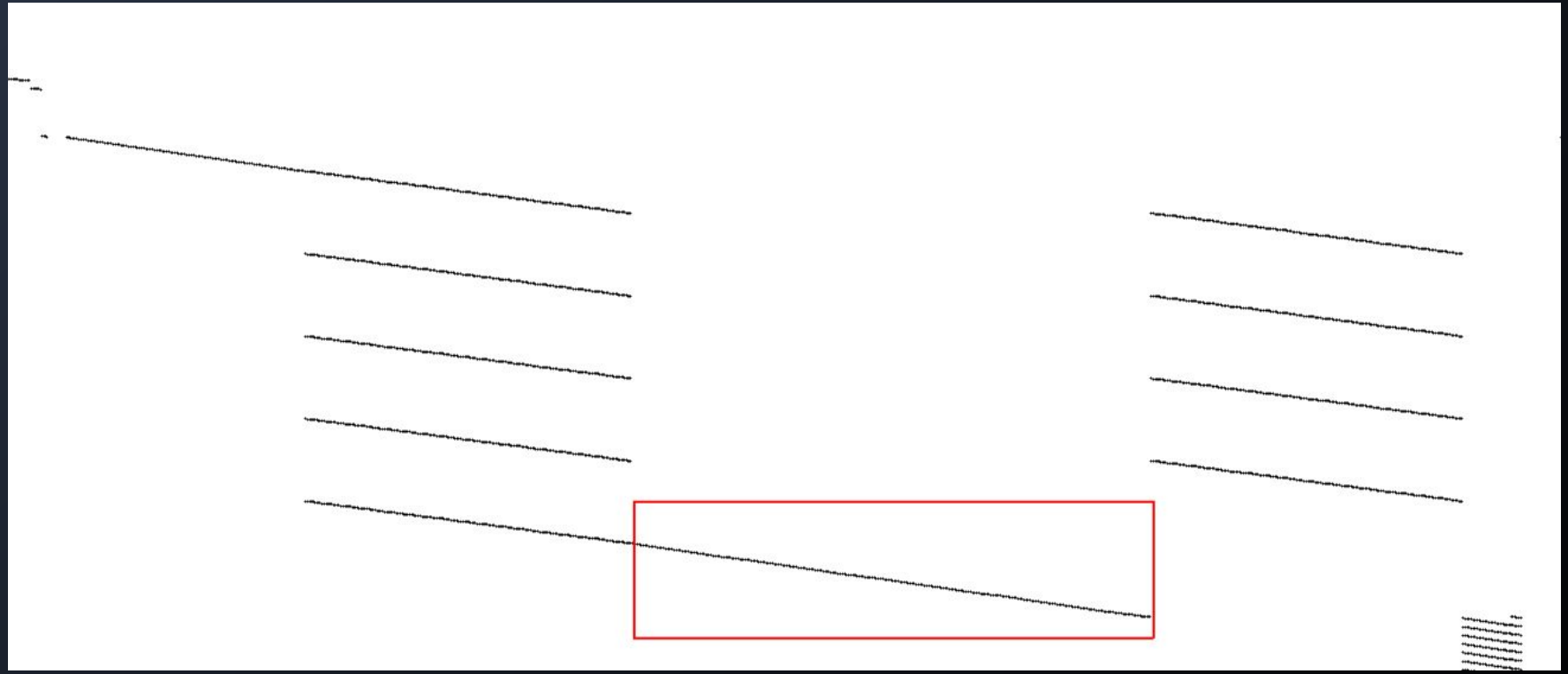
    AddRoundKey

Substitution

ShiftRows

AddRoundKey

Finding the Fault Position - AES



Last round

AES DFA in 5 Simple Steps

✓ 1. Identify the AES operations
2. Inject a fault at a precise location that corrupts the internal state
3. Generate many faulted outputs for the same input
4. Throw the result into phoenixAES
5. Profit?

AES DFA in 5 Simple Steps

✓ 1. Identify the AES operations

2. Inject a fault at a precise location that corrupts the internal state

3. Generate many faulted outputs for the same input

4. Throw the result into phoenixAES

5. Profit?

AES DFA in 5 Simple Steps

✓ 1. Identify the AES operations

2. Inject a fault at a precise location that corrupts the internal state

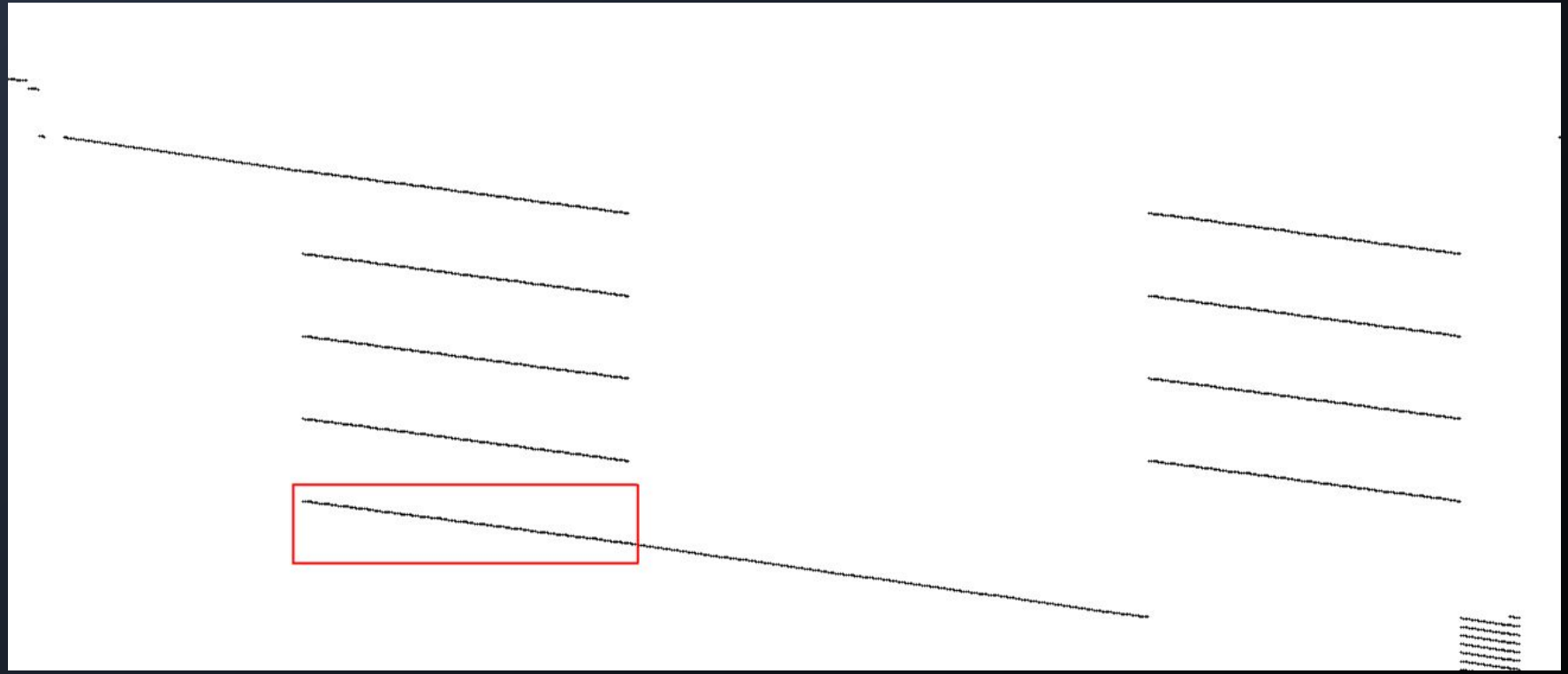
3. Generate many faulted outputs for the same input

4. Throw the result into phoenixAES

5. Profit?

Needs to be between the last two MixColumn operations

Finding the Fault Position - AES



Estimated fault position

AES DFA

If all bytes change, the fault is introduced too early!

```
FAULT TARGET: 0x75e
EXP: 63616161616161616161616161617961
GOT: 86DF0153F2AE873342C62AF241EBDF5C
FAULT TARGET: 0x75f
EXP: 63616161616161616161616161617961
GOT: 96FAE82DC0166648E5CB4BB081E7A3A3
FAULT TARGET: 0x760
FAULT TARGET: 0x761
EXP: 63616161616161616161616161617961
GOT: 3B61616161B461616161F1616161796C
Total count: 1
FAULT TARGET: 0x762
FAULT TARGET: 0x763
FAULT TARGET: 0x764
FAULT TARGET: 0x765
EXP: 63616161616161616161616161617961
GOT: F261616161436161616114616161793E
Total count: 2
FAULT TARGET: 0x766
```

To introduce the fault, we just skip an instruction.

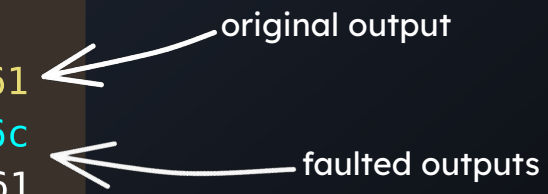
The position is correct, when after introducing the fault exactly 4 bytes change in the output for the same input.

Nd

AES DFA

tracefile

```
1 63616161616161616161616161617961
2 3b61616161b461616161f1616161796c
3 6361614 3c0616161616b6161616164 61
4 ...
```




```python
import phoenixAES

print(phoenixAES.crack_file("tracefile", encrypt=False))
```

AES DFA

```python
key = bytes.fromhex("<phoenixAES output>")

with open("rootfs/data/vendor/mediadrm/IDM1013/L3/ay64.dat" , "rb") as f:
    data = f.read()

keybox = AES.new(key, AES.MODE_CBC, iv=b"\x00" * 16).decrypt(data)

_device_id, _device_key, _data, magic, _crc = struct.unpack(
    "<32s16s72sII", keybox
)

assert magic == struct.unpack("<I", b"kbox")[0], "Keybox magic mismatch"

print("Successfully decrypted keybox:")
print(keybox.hex())
```

Decrypt the Keybox from Storage

Recap

1. Create an execution trace
2. Find the AES code using TraceGraph
3. Inject faults between the last two MixColumns operations
4. Write the faulty outputs to a file
5. Use phoenixAES to recover the encryption key
6. Decrypt ay64.dat to get the keybox

The Widevine Playground

A

Key extraction using Differential Fault Analysis

B

Reverse engineering the obfuscation

C

Dumping the keybox from memory

Nd Neodyme

05

Reverse Engineering

Reverse Engineering

The Initialization function _Icc01 creates a VM and sets up a vendor specific key.

The VM has handlers for platform dependent operations (filesystem accesses, device name, …).

The VM code is encrypted.

Reverse Engineering

VM Code Decryption

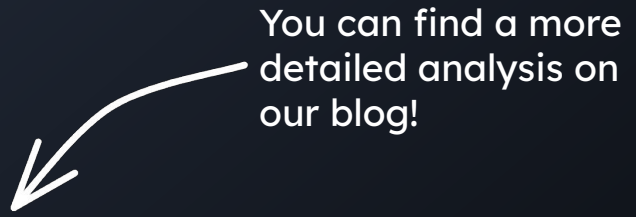
Each function has its own encryption key and a checksum for the decrypted code.

We can just hook the code that verifies the checksum to dump the decrypted functions at runtime to a file.

The code flow is indirect, but we recover it from a trace.

Reverse Engineering

Code reconstruction



The code is not too hard to reconstruct at this point.

- Take shortcuts by using FindCrypt
- Reconstruct the keybox generation and encryption
  - Keybox encryption key is just SHA1 hash of the device unique id

Reverse Engineering

Code reconstruction

You can find a more detailed analysis on our blog!

The code is not too hard to reconstruct at this point.

- Take shortcuts by using FindCrypt
- Reconstruct the keybox generation and encryption
  - Keybox encryption key is just SHA1 hash of the device unique id

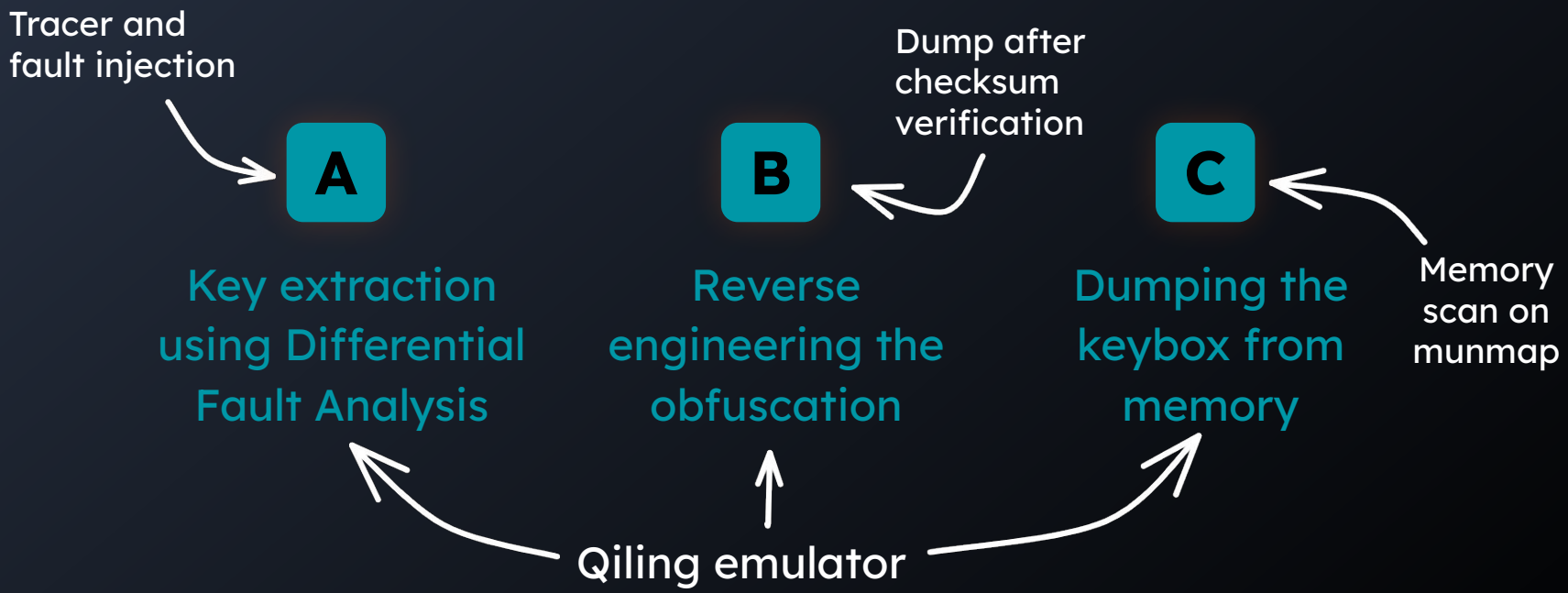
This actually has not changed in newer versions

Generate the Keybox from Scratch

Recap

1. Reverse engineer the Keybox generation
2. Create a custom implementation
3. Request a new device certificate from the Widevine servers
4. Request a license for the media you want to play
5. Obtain the media decryption key using your RSA key
6. Decrypt the media

The Widevine Playground



Nd Neodyme

Want to try it yourself? The entire code is available on GitHub!

06

Now it’s your turn!

You can find the blogpost over here!

Contact: @_localo_
Email: felipe@neodyme.io
Blog: blog.neodyme.io