Dynamic Binary Instrumentation Techniques to Address Native Code Obfuscation

Romain Thomas
rthomas@quarkslab.com

Abstract

Android applications are becoming more and more obfuscated to prevent reverse engineering. While obfuscation can be applied on both the Dalvik bytecode and the native code, the former is more challenging to analyze due to the structure of the bytecode as well as the API provided by Android Runtime.[1]

The purpose of this talk is to present dynamic binary instrumentation techniques that can help reverse engineers to deal with obfuscated codes. These techniques aim to be obfuscator resilient so that it does not rely on a special kind of obfuscation nor a specific obfuscator.

Keywords: Android, Obfuscation, ARM, AArch64, Reverse Engineering, Instrumentation

Introduction

Android applications embed more and more critical assets that must be protected from reverse engineering. These assets mostly depend on the purpose of the application, and how they are used within the application. We highlighted three different categories of assets that are prone to reverse engineering, and thus, obfuscation:

- **Protocol**: message structures, endpoints, signature, …
- **Secret keys**: API Token, certificates, RSA keys, …
- **Algorithms**: anti-tampering, integrity checks, whiteboxes, anti-root, …

As an example, social media and messaging applications may use obfuscation to avoid third-party clients that would not be under control of the application's owners.

In the bank industry, security standards require the use of obfuscation as a mandatory step in the development process. Thus, to assess the security of these applications, analysts usually have to deal with several layers of protection.

Regarding video games, we encountered obfuscation to prevent cheat, bots as well as protections for the in-app billing capabilities.

Finally, most of the DRM solutions are protected through obfuscation, even though the current trend is to use a secure element such as TrustZone.

The next parts present **D**ynamic **B**inary **I**nstrumentation techniques that aim to extract relevant information from the noise introduced by the obfuscators. These techniques target different kinds of dynamic information depending on the purpose of the obfuscated code. Among this information, we will describe the process to extract:

- Call trace of internal and external functions (e.g those from `libc.so` or `libart.so`)
- Call trace of JNI functions (e.g. `NewString`, `CallObjectMethod`, `RegisterNatives`)
- Memory trace

[1] `http://androidxref.com/8.1.0_r33/xref/art/runtime/instrumentation.h#61`

- Instruction trace

While dealing with obfuscated code is a challenge in itself, we faced another one in the enhancement of QBDI to support the `ARM` and `AArch64` instruction sets (which includes Thumb and Thumb2).

1. Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (**DBI**) is an analysis technique that aims to observe program's behavior at different levels:

- Instructions: by providing callback before or after the instructions.
- Basic Block: by triggering an event when a basic block is executed or when a basic block is discovered (e.g. code coverage)
- Memory instrumentation: by monitoring memory addresses and values used by the program

It can be seen as an enhanced debugger without breakpoints and with real-time performances.

In the case of obfuscated code, a DBI turns out to be quite efficient as explained in the publications and different papers over the last 5 years[1], [2], [3], [4], [8].

2. Current state of DBI frameworks

There are several DBI frameworks, each one addressing the problem in a different way. Among these frameworks, we can find:

Intel PIN which is reliable on the `x86` and `x86-64` architectures but it does not support `ARM` or `AArch64`. It might also have some issues to run a program that can't be linked with Intel PIN's CRT.

Valgrind that can instrument code running on `x86`, `x86-64`, `ARM` and `AArch64` but the project is not very modular to be smoothly integrated with other frameworks. The API may also be tricky to use as well as the compilation for Android.

DynamoRio which is mostly used on Windows but also supports ARM and AArch64. Nevertheless, the support on ARM and AArch64 is limited[2].

DynInst *- Not tested*

Frida that recently released an ARM and AArch64 version of its *stalker* which basically enables to trace instructions. Its implementation uses the stack[3] and the API is very user-friendly.

Depending on what we are looking for and the environment on which the target is executed, one of these DBI frameworks may be more convenient than another.

In the case of **Android applications** running on **ARM** or **AArch64**, only Frida seems to be able to address the reverse engineering problems.

3. QBDI: Introduction and Techniques to Handle Obfuscation

QBDI [9] is a cross-platform and cross-architecture DBI created by C. Hubain [4] and C. Tessier, two reverse engineers with a strong background reverse-engineering and obfuscation.

It is based on LLVM and has been designed with a modular architecture so that it can be combined and integrated with other tools like Frida. LLVM provides the two main components that make the instrumentation process: a disassembler (`llvm::MCDisa`) and an assembler (`llvm::MCCodeEmitter`). In addition, LLVM provides a handy abstraction (`llvm::MCInst`, `llvm::MCInstrDesc`) over the underlying assembly instruction.

As a must-have feature for DBI frameworks, QBDI enables to setup callbacks before or after instructions so that users can inspect the context — like CPU registers — in which the instruction is executed. The

[2] `https://github.com/DynamoRIO/dynamorio/wiki/AArch64-Port#stolen-register`

[3] Which can be used to detect or break the stalker

instrumentation process is summarized in figure 1. The left-hand side represents the instructions to be instrumented and the right-hand side, the logical view of the QBDI's callbacks.

```
0x17505A LDR    R0, [R5]
0x17505C MOV    R8, R1
0x17505E LDR    R4, [R1, R6]
0x175060 LDR    R2, [R0, #0x18]
0x175062 MOV    R0, R5
0x175064 MOV    R1, R4
0x175066 BLX    R2
0x175068 MOV    R1, R0
0x175068 MOV    R1, R7
0x17506A MOV    R0, R7
0x17506C SVC    #0
0x17506E MOV    R1, R0
0x175070 AND.W  R1, R0, R3
0x175074 EOR.W  R0, R0, R3
0x175078 BLX    sub_175094
```

```
QBDI                Instruction Callback
.text:0x17505A  LDR    R0, [R5]
QBDI                Instruction Callback
.text:0x17505C  MOV    R8, R1
QBDI                Instruction Callback
.text:0x17505E  LDR    R4, [R1, R6]
QBDI                Instruction Callback
.text:0x175060  LDR    R2, [R0, #0x18]
QBDI                Instruction Callback
.text:0x175062  MOV    R0, R5
QBDI                Instruction Callback
.text:0x175064  MOV    R1, R4
QBDI                Instruction Callback
.text:0x175066  BLX    R2
QBDI                Instruction Callback
.text:0x175068  MOV    R1, R0
QBDI                Instruction Callback
.text:0x175068  MOV    R1, R7
QBDI                Instruction Callback
.text:0x17506A  MOV    R0, R7
QBDI                Instruction Callback
.text:0x17506C  SVC    #0
QBDI                Instruction Callback
.text:0x17506E  MOV    R1, R0
QBDI                Instruction Callback
.text:0x175070  AND.W  R1, R0, R3
QBDI                Instruction Callback
.text:0x175074  EOR.W  R0, R0, R3
QBDI                Instruction Callback
.text:0x175078  BLX    sub_175094
```

Figure 1 – Callback instrumentation

Programmatically, it is achieved with the following API:

```
VMAction callback(VMInstanceRef vm,
                  GPRState *gprState, FPRState *fprState,
                  /* user data */ void* ctx) {
  ...
  return VMAction::CONTINUE;
}

// ==================================================== //

QBDI::VM vm;

vm.addCodeCB(QBDI::InstPosition::PREINST, callback,
             /* user data */&ctx);

vm.call( ... , function, fnc_args);
```

This kind of instrumentation could be used to generate an instruction trace that can then be analyzed with other tools such as Triton[11], Scared[12] or Daredevil[13].

Nevertheless, instrumenting all the instructions can add a significant overhead, and the output size can be huge and time-consuming to process.

To address this issue, QBDI enables to create *rules* to select what kind of instructions aims to be instrumented. For instance, one can choose to only instrument syscalls or instructions that perform memory accesses. More precisely, QBDI exposes a rules engine that can filter instructions depending on their semantics. This rules engine relies on the `llvm::MCInstrDesc` interface to provide an unified filters regardless of the underlying architecture.

For instance, one can filter instructions based on the following properties:

- **Mnemonics**: `bl`, `add`, `and`, ...
- **Categories**: syscall: `SVC`, system instructions: `MSR`, `MRS`, `HVC`, ...
- **Properties**: memory access, read memory access, call (`llvm::MCInstrDesc.isCall`) ...

As obfuscators tend to add noisy instructions that would not be relevant for analysts, we based our analysis on the following rules that match a good trade-off between the trace's size and the relevance of the information generated for the reverse-engineering:

- Syscalls
- Calls
- Memory Accesses

The figure 2 shows the instrumentation process based on these rules.

```
0x17505A  LDR    R0, [R5]
0x17505C  MOV    R8, R1
0x17505E  LDR    R4, [R1, R6]
0x175060  LDR    R2, [R0, #0x18]
0x175062  MOV    R0, R5
0x175064  MOV    R1, R4
0x175066  BLX    R2
0x175068  MOV    R1, R0
0x175068  MOV    R1, R7
0x17506A  MOV    R0, R7
0x17506C  SVC    #0
0x17506E  MOV    R1, R0
0x175070  AND.W  R1, R0, R3
0x175074  EOR.W  R0, R0, R3
0x175078  BLX    sub_175094
```

```
QBDI                 Memory Callback
.text:0x17505A  LDR    R0, [R5]
.text:0x17505C  MOV    R8, R1
QBDI                 Memory Callback
.text:0x17505E  LDR    R4, [R1, R6]
QBDI                 Memory Callback
.text:0x175060  LDR    R2, [R0, #0x18]
.text:0x175062  MOV    R0, R5
.text:0x175064  MOV    R1, R4
QBDI                 Call Callback
.text:0x175066  BLX    R2
.text:0x175068  MOV    R1, R0
.text:0x175068  MOV    R1, R7
.text:0x17506A  MOV    R0, R7
QBDI                 Syscall Callback
.text:0x17506C  SVC    #0
.text:0x17506E  MOV    R1, R0
.text:0x175070  AND.W  R1, R0, R3
.text:0x175074  EOR.W  R0, R0, R3
QBDI                 Call Callback
.text:0x175078  BLX    sub_175094
```

Figure 2 – Instrumentation based on rules

The following code is an example of the QBDI's API to select a subset of instructions to instrument:

```
QBDI::VM vm;

// Callbacks before syscall instructions
vm.addSyscallCB(PRE_SYSCALL, syscall_cbk, &ctx);

// Callbacks before blx / bl instructions
vm.addCallCB(PRE_CALL, call_cbk, &ctx);

// Callbacks before memory load and store instructions
vm.addMemAccessCB(MEMORY_READ_WRITE, mem_cbk, &ctx);

vm.call( ... , function, fnc_args);
```

Call Resolution

In addition to instrumentation rules, QBDI enables to resolve the address of calls instruction (e.g `blx`, `br`, `bl`) so that users can access to the effective address being called.

```
VMAction call_cbk(VMInstanceRef vm, ... ) {

  // Provide:
  // BLX R2  -> Value of R2
  // BLX PC  -> PC + CPU Mode shift + Alignment
  // BLX #42 -> PC + 42 + CPU Mode shift + Alignment
  uintptr_t call_target = vm->getInstCallAccess().back();

  ...
  return VMAction::CONTINUE;
}
```

This resolution is particularly useful in the case of indirect calls such as `blx r3`. Obfuscators are prone inserting this kind of instruction, as they usually break static analysis.

Dynamically, the call target is straightforward to retrieve since we have access to the register values. The figure 3 shows an example of the static and dynamic outputs.

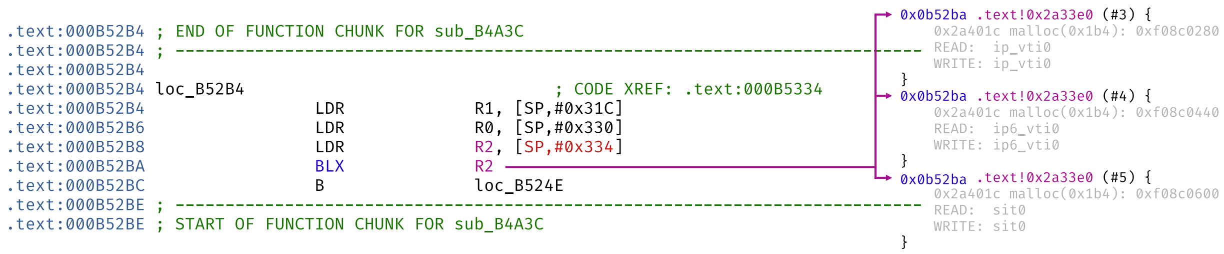


Figure 3 – Dynamic vs static output

The advantage of the API provided by QBDI is that the user does not have to process the call instruction to determine which register is used and which absolute address is called. In addition, the ARM Thumb/Thumb2 architecture performs *implicit* alignments that can be annoying to handle manually.

Given the absolute call address, we can post-process this value to improve its meaning. While `call 0x7fbc2a33e0` is not really meaningful for the analyst, the transformation to: `library:section!offset or symbol` makes more sense.

Because QBDI is injected in the **same memory space** as the target to analyze, we have access to the memory layout which contains the base addresses and the libraries paths.

Memory Accesses Resolution

Similarly to call resolution, QBDI can resolve the value and the **effective memory address**. It means that the memory address being read or written is provided within the callback so that users can access this information without looking at the underlying instruction.

```
VMAction mem_cbk(VMInstanceRef vm, ... ) {

  // Memory info
  MemoryAccess info = vm->getInstMemoryAccess().back();
  // Provide:
  // LDR  R0, [R5]               -> R5
  // LDR  R4, [R1, R6]           -> R1 + R6
  // LDR  R2, [R3, #0x18]        -> R3 + 0x18
  // STRB R8, [R1, -R2, LSL #4] -> R1 - (R2 << 4)
  uintptr_t addr = maccess.accessAddress;

  // Value read or written (R0, R4, R2, R8)
  uintptr_t value = maccess.value;
  ...
  return VMAction::CONTINUE;
}
```

The `MemoryAccess.accessAddress` attribute[4] is filled with the address according to the addressing mode. For instance, on the instruction `STRB R8, [R1, -R2, LSL #4]`, it will contain the value of $\mathtt{R1}_{value} - (\mathtt{R2}_{value} \times 16)$. As for call resolution, this feature is abstracted to the user so that the callbacks that use this interface can work regardless of the underlying architecture (`x86` vs `AArch64`).

One could use this `MemoryAccess` interface, to track **bytes** memory accesses within a function and filter on the printable values. Such a heuristic is quite efficient to locate a string decoding routine. Even though strings are statically protected by the obfuscators, at some point in the execution they need to be decoded. In most cases, the decoded string is stored into a memory buffer which implies **write** memory accesses. By tracking these accesses and inspecting the values (`MemoryAccess.value`), we are likely to observe the clear strings no matter how the complexity of the string transformation is:

```
-----> Enter in sub_1421c <-----
0x014274 .text!0x177c (#0) {
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29240).b: 0x1e
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af70).b: /
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29241).b: 0xeb
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af71).b: s
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29242).b: 5
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af72).b: y
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29243).b: 0xd5
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af73).b: s
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29244).b: '
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af74).b: t
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29245).b: 0xcc
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af75).b: e
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29246).b: 9
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af76).b: m
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29247).b: 0x85
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af77).b: /
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29248).b: 7
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af78).b: b
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x29249).b: C
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af79).b: i
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x2924a).b: {
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af7a).b: n
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x2924b).b: %
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af7b).b: /
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x2924c).b: v
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af7c).b: s
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x2924d).b: w
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af7d).b: u
  0x00179c  ldrb  w4, [x2, x5] -> [R] (.data + 0x2924e).b: 0x1
  0x0017ac  strb  w4, [x0, x5] -> [W] (.bss  + 0x2af7e).b: 0x0
}
0x014288 .text!0x16c80 (#0) {
  0x016ca0 openat('', '/system/bin/su')
}
```

The figure 4 outlines the process.

[4] https://github.com/QBDI/QBDI/blob/39a936b2efd000f0c5def0a8ea27538d7d5fab47/include/QBDI/Callback.h#L130

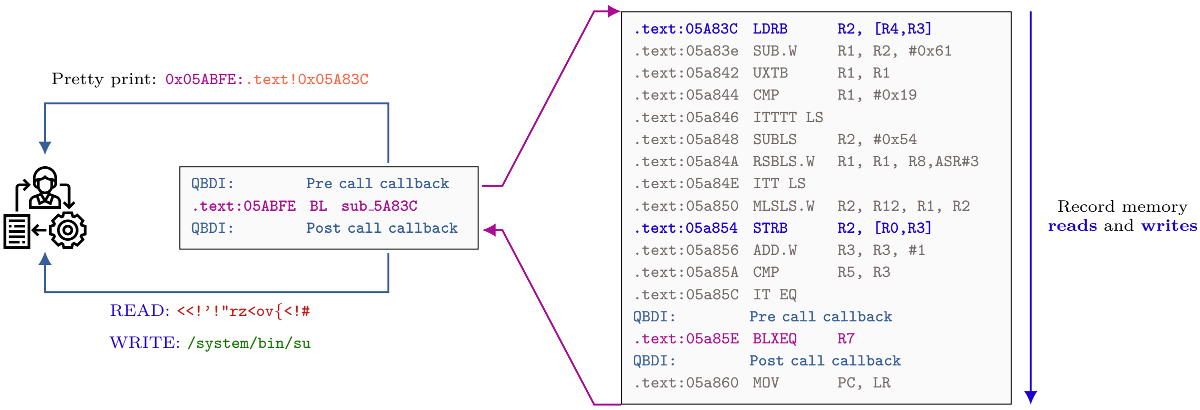


Figure 4 – Memory and call instrumentation to detect string decoding routine

One can find additional information on the Quarsklab blog: *Android Native Library Analysis with QBDI - Encoding Routine*[5].

The ExecBroker

Instrumented code is likely to call external functions like `malloc()`, `mmap()` or `env->FindClass()` whose the instrumentation[6] would not be relevant — from a reverse-engineering point of view — to understand the logic of the obfuscated code. Moreover, these external functions may use shared variables[7] with QBDI that could lead to deadlock or infinite loops.

This limitation is well known by the DBI frameworks and Intel PIN choose to address this issue to provide its own `C & C++` runtime [8]. On the other hand, QBDI implements a different mechanism that stops the instrumentation process when an external call is detected and resumes the process when the function finishes.

The *ExecBroker* is the QBDI's component that implements this mechanism. The figure 5 represents the different events when an external call occurs during the instrumentation.

[5] `https://blog.quarkslab.com/android-native-library-analysis-with-qbdi.html#encoding-routine`
[6] e.g. instruction trace
[7] mutex, static variables, ...
[8] `https://software.intel.com/sites/default/files/managed/8e/f5/PinCRT.pdf`

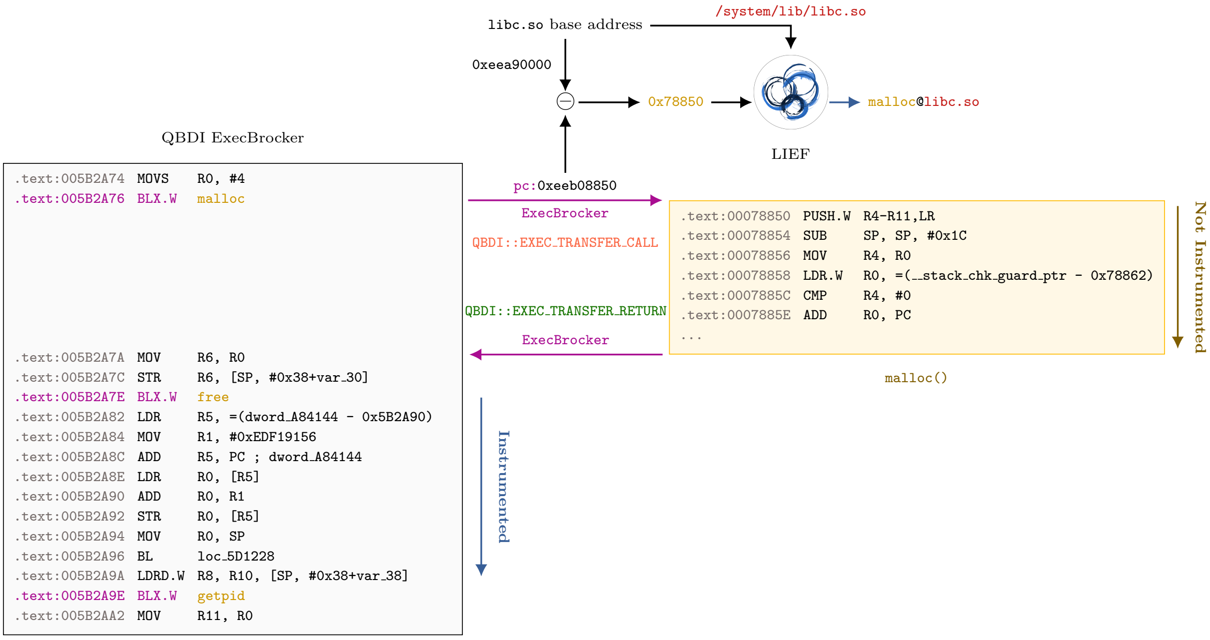


Figure 5 – Instrumentation based on rules

At the address `0x5B2A76`, QBDI detects a call to the absolute address `0xeeb08850` that is not included in the instrumentation ranges: it's considered as an external call. Therefore, QBDI transfers its CPU internal representation into the *real* CPU and changes the return address — `lr` register — to a special value so it can catch when the function returns. The function `0xeeb08850` is then executed by itself, without instrumentation.

When QBDI performs these operations, it informs the user by triggering two events:

- `QBDI::EXEC_TRANSFER_CALL`
- `QBDI::EXEC_TRANSFER_RETURN`

The event `QBDI::EXEC_TRANSFER_CALL` is triggered before running the function without instrumentation while `QBDI::EXEC_TRANSFER_RETURN` is generated when the function finished its execution.

To convert the absolute address `0xeeb08850` into a symbol, one can first detect the module in which the address is located. This step can be done by iterating on `/proc/self/maps`. Then, we can subtract the module base address from `0xeeb08850` to get a relative offset within the library.

Finally, using the library path `/system/lib/libc.so` and the offset `0x78850`, we can use an ELF parser like LIEF[10] to resolve the offset into a symbol name.

In the figure 5, the address `0xeeb08850` is resolved into `malloc()`. One can find a small example of this conversion in the QBDI's examples[9]

This association between QBDI and LIEF can be used to generate a call trace of external functions. Furthermore, as we are able to resolve calls into symbols, we can specialize the QBDI callbacks to handle and pretty print the function's parameters:

[9] https://github.com/QBDI/examples/blob/master/packer-android-x86/src/libshellx_qbdi.cpp#L18-L50

```
VMAction exec_broker(VMInstanceRef, const VMState *vmState, GPRState *gprState, FPRState*, void* ctx) {

  std::string symbol = resolve(gprState->pc);

  if ((vmState->event & QBDI::EXEC_TRANSFER_CALL) != 0) {
    if (symbol == "malloc") {
      const size_t malloc_size = gprState->x0;
    }
  }

  if ((vmState->event & QBDI::EXEC_TRANSFER_RETURN) != 0) {
    if (symbol == "malloc") {
      const uintptr_t malloc_addr = gprState->x0;
    }
  }

  return QBDI::CONTINUE;
}

vm.addVMEventCB(EXEC_TRANSFER_CALL,   exec_broker, ctx);
vm.addVMEventCB(EXEC_TRANSFER_RETURN, exec_broker, ctx);
```

Figure 6 – ExecBroker parameters and return value processing

Compared to Frida hooking, the ExecBroker enables to trace external functions *a priori* so that the user does not have to setup *hook* beforehand. In addition, QBDI doesn't modify the assembly code: it just changes the return address. Therefore, checking the integrity of `/system/lib64/libc.so` is not efficient to detect QBDI's ExecBroker while it is to detect Frida[10]. On the other hand, the instruction that performs the external call needs to be in the instrumented range while Frida enables to catch the call unconditionally.

One can also use the ExecBroker to track functions that perform dynamic memory allocations (`malloc`, `mmap`, `...`) and inspect the memory buffers when they are released (e.g. with `free()`). Thanks to `QBDI::EXEC_TRANSFER_CALL`, we can access the allocation's size and with `QBDI::EXEC_TRANSFER_RETURN` we can access the allocated address (see 6). These values (size and allocated address) can be stored in a *context* structure so that when the buffer is released we know exactly its size. We can then iterate over its bytes which could reveal strings or identifiers.

Figure 7 shows an example of this technique on the Tencent's packer. On the left-hand side, we detect an allocation of `0x819358` bytes located at address `0x7e33200000`. Later in the execution, we detect that this address is freed and by inspecting the buffer, we can find magic bytes of DEX file[11].

```
0x00f0b4 .text!0x173fc (#0) {
  0x01742c malloc(0x819358): 0x7e33200000
  0x017444 .text!0x170b4 (#0) {
    0x0170d8 puts('~~~~~~~~~~~~~')
    0x0170f4 printf([ucl_nrv2d_decompress_8]src=%x,src_len=%x,dst=%x,dst_len=%x)
  }
}
0x00f0b4 .text!0x173fc (#1) {
  0x01742c malloc(0x8ac734): 0x7e32800000
  0x017444 .text!0x170b4 (#1) {
    0x0170d8 puts('~~~~~~~~~~~~~')
    0x0170f4 printf([ucl_nrv2d_decompress_8]src=%x,src_len=%x,dst=%x,dst_len=%x)
  }
}
0x00f0b4 .text!0x173fc (#2) {
  0x01742c malloc(0x8a9124): 0x7e31e00000
  0x017444 .text!0x170b4 (#2) {
    0x0170d8 puts('~~~~~~~~~~~~~')
    0x0170f4 printf([ucl_nrv2d_decompress_8]src=%x,src_len=%x,dst=%x,dst_len=%x)
  }
}
```

```
0x00e1dc env->NewStringUTF('UnpoisonDexDataToMemory:end'): ...
 ...
0x008a38 free(0x7e33200000): HWdex035XS>_eA'-H`jHpxV4lp'$7z((%i~XXX
0x008a38 free(0x7e32800000): $Wdex035f&O,g|(\$pxV4H;p+\e6>=8_"7?od]
0x008a38 free(0x7e31e00000): XWdex035}l#>pxV48&p'5(C'qxXX)XXXvXXXX"
```

Figure 7 – Memory allocation in Tencent's packer

4. Uses Cases

The previous sections introduced QBDI's features that can be used to instrument code. The next sections expose four use cases on obfuscated code from different obfuscators.

[10]See the challenge `R2pay.apk` released in the r2con CTF 2020

[11]It has been confirmed with a manual analysis but the DEX file was somehow truncated

4.1 `JNI_OnLoad` Obfuscation

The Java language specification lets developers declare functions whose the implementation is located in a native library. The definitions of these functions use the keyword `native` as shown in the figure 8.

```java
package gh;

public class wer {
  private static final byte[] e = {41, 82, -31, 109, 9, 85, 95, 77, 57, 121, -53, 255, 59, -70, ...};

  public static native String a(String[] strArr, String[] strArr2, String str, byte[] bArr);

  public static native String b(String str);

  public static native String c(String str);

  public static native int d(byte[] bArr, byte[] bArr2);
}
```

Figure 8 – JNI functions in a Java class

In terms of obfuscation, this technique is interesting since it moves the logic of the functions in native code that can be more efficiently obfuscated than the Dalvik Bytecode.

In this kind of protection, we usually find the entrypoint of the library in the `JNI_OnLoad` [12] function that aims to bridge native functions declared in the Java code with a pointer in the library.

To figure out the offsets of the JNI functions, reverse engineers need to identify the external call to `env->RegisterNatives()` — which is exposed by the Android runtime — and inspect the function parameters to find the `JNINativeMethod` structure that contains the offsets of the JNI functions.

Using the QBDI's ExecBroker, we can dynamically catch the call to `RegisterNatives()` and setup a callback that inspects the parameters. In particular, the second parameter: `r2`/`x2`, points to the `JNINativeMethod` structure.

The figure 9 shows the control-flow graph of the obfuscated `JNI_Onload()` function in Snapchat and the figure 10 shows the output of QBDI on this function.

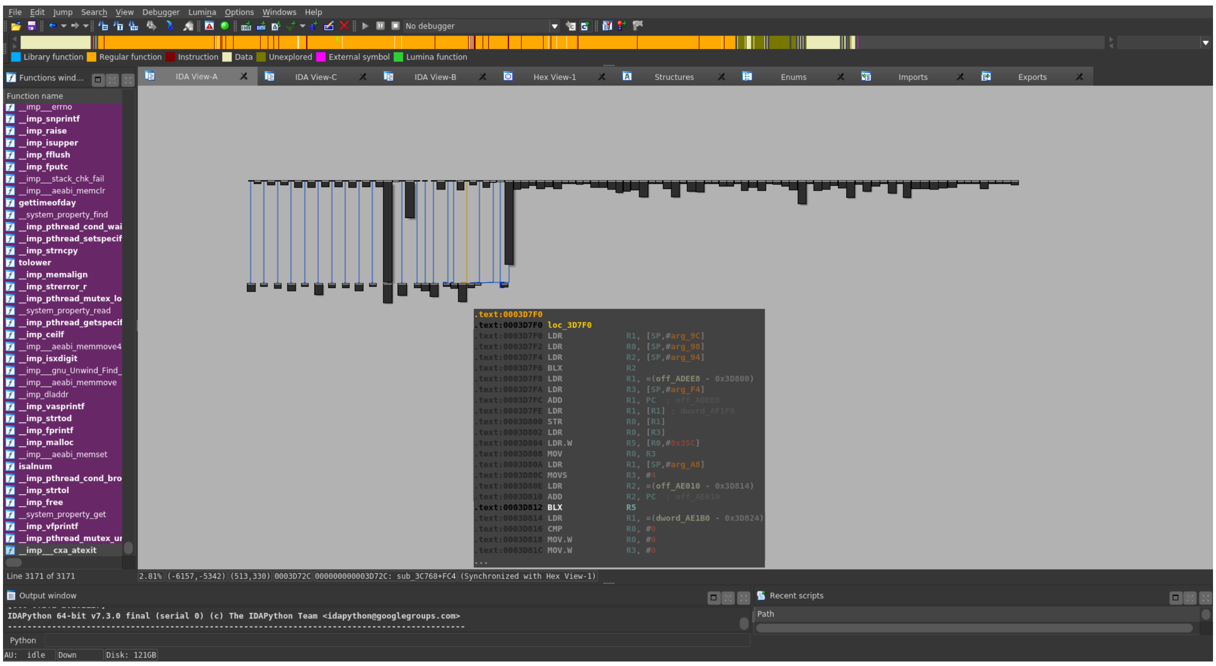


Figure 9 – Obfuscated CFG of Snapchat's `JNI_OnLoad`

[12] It exists another way to expose these functions through a special naming (`Java_<class>_<method>`) but it leaks the symbol and its offset.

```
0×0248fe .text!0×3c769 (#0) {
  0×03d5b4 jvm→GetEnv( ... )
  0×03d640 env→FindClass('gh/wer'): 0×5
  0×03d748 env→FindClass('java/lang/RuntimeException'): 0×19
  0×03d7f6 env→NewGlobalRef(0×19): 0×ee6
  0×03d812 env→RegisterNatives('gh/wer', ... , nb_methods=4)
  a → /data/local/tmp/libXYplugin.10.59.0.0.so@0×24f85
  b → /data/local/tmp/libXYplugin.10.59.0.0.so@0×24629
  c → /data/local/tmp/libXYplugin.10.59.0.0.so@0×27251
  d → /data/local/tmp/libXYplugin.10.59.0.0.so@0×84e15
  0×03d8ce .text!0×37695 (#0) {}
  0×03d900 .text!0×4c629 (#0) {
      0×04c878 .text!0×42b9d (#0) {
          0×047744 .text!0×52bb9 (#0) {
              0×054a90 env→FindClass('com/XXXXXXXX/android/framework/misc/AppContext'): 0×21
          }
          0×047756 env→NewGlobalRef(0×21): 0×ef6
          0×047744 .text!0×52bb9 (#1) {
              0×054a90 env→FindClass('android/content/Context'): 0×25
          }
     ...
  }
```

Figure 10 – QBDI Output

From this output, we can quickly identify the location of `gh.wer.a()` which is at the offset `0x24f85` in the library.

This technique to resolve JNI functions is generic and could be applied in other applications whatever the underlying obfuscation.

4.2 Android Packer

In addition to native code obfuscation, applications can use packers to add another layer of protection.

During our experimentations, we dealt with an application protected by a commercial packer[13]. This solution uses several layers of protection that are tedious to analyze statically. Moreover, it implements different anti-debug to prevent dynamic analysis.

The next sections outline some of these protections.

4.2 Watermarking

The main part of the protection is located in a native library named `libXYZprotector.<pid>.so`. By tracing the library with QBDI[14], we noticed a particular sequence of calls that are represented in the figure 11.

```
0×00514c .text!0×6cd6 (#0) {
  0×006ce0 stat64('/data/local/tmp/lib<protector>.so')
}
0×0052c8 .text!0×6c12 (#1) {
  0×006c18 .text!0×6be0 (#2) {
  }
  0×006c20 open('/data/local/tmp/lib<protector>.so')
}
0×005228 .text!0×6bb8 (#0) {
  0×000000 lseek(9, 0×0008, 0)
}
0×005316 .text!0×6b92 (#575) {
  0×006ba0 read(9, 0×ffcddabc, 0×8): DPLF
}
0×00517a .text!0×6be0 (#3) {
  0×006bea __errno()
  0×006bf2 close()
  0×006c04 __errno()
}
```

Figure 11 – Dynamic trace generated with QBDI

[13] which is not Legu
[14] Configured with the ExecBroker, syscall & call callbacks

From this trace, we can see that the function performs the following actions:

1. open the raw library: `open(...)`
2. seek and read the ELF's identity field
3. verify the field's value (*not shown*)

By comparing the ELF identity field between a genuine library and the one from the packer, we identified a difference that is emphasized in figure 12.

```
In [1]: a = lief.parse("/bin/ls")

In [2]: b = lief.parse("lib<protector>.so")

In [3]: print(a.header.identity)
[127, 69, 76, 70, 01, 01, 01, 00, 00, 00, 00, 00, 00, 00, 00, 00]
                                  Genuine
In [4]: # -ELF------------

In [5]: print(b.header.identity)
[127, 69, 76, 70, 01, 01, 01, 00, 68, 80, 76, 70, 00, 224, 00, 00]
                                  Watermark
In [6]: # -ELF---DPLF-----
```

Figure 12 – ELF identity

This difference strongly suggests that the packer uses the padding area of the ELF identity field to watermark the library. Moreover, this modification breaks the Linux's loader when trying to load the `x86-64` version on Linux[15].

4.2 Anti dump & Anti Debug

To prevent a memory dump that could be used to extract the in-memory DEX files, the packer implements classical anti-dump and anti-debug techniques. These protections are not new but they are wrapped with different layers of obfuscations.

The figure 13 shows the basic block and the CFG of the function involved in the anti-debug and the anti-dump. Once the basic block identified, it is straightforward to understand its logic and how it protects the application against dump and debugging. The main difficulty is to identify the basic block among those that are melt in the function.

[15] `https://blog.quarkslab.com/when-sidechannelmarvels-meet-lief.html#converting-an-android-library-to-linux-with-lief`

```
LOAD:000451D0 MOV  W3, W1
LOAD:000451D4 LDR  X9, [X9]
LOAD:000451D8 MOV  W19, W4
LOAD:000451DC MOV  X25, X0
LOAD:000451E0 MOV  W4, W1
LOAD:000451E4 MOV  W0, #4
LOAD:000451E8 STR  X9, [SP,#0x428]
LOAD:000451EC STR  X5, [SP,#0xF8]
LOAD:000451F0 ADD  X20, SP, #0x158
LOAD:000451F4 STR  X6, [SP,#0xF0]
LOAD:000451F8 ADD  X23, SP, #0x1A8
LOAD:000451FC STR  X7, [SP,#0xE8]
LOAD:00045200 BL   prctl
LOAD:00045204 ADD  X2, SP, #0x128
LOAD:00045208 MOV  X3, #0xFFFFFFFFFFFFFFFF
LOAD:0004520C MOV  X1, X2
LOAD:00045210 MOV  W0, #4
LOAD:00045214 STR  X3, [SP,#0x430+var_300]
LOAD:00045218 ADRP X24, #0x74010@PAGE
LOAD:0004521C STR  XZR, [SP,#0x430+var_308]
LOAD:00045220 BL   setrlimit
LOAD:00045224 BL   getpid
LOAD:00045228 MOV  W1, W0
LOAD:0004522C MOV  W2, #0
LOAD:00045230 MOV  W0, #0x6D61
LOAD:00045234 MOV  W3, W2
LOAD:00045238 MOV  W4, W2
LOAD:0004523C MOVK W0, #0x5961,LSL#16
LOAD:00045240 BL   prctl
LOAD:00045244 STR  W21, [X20]
LOAD:00045248 LDR  W1, [SP,#0x430+arg_0]
LOAD:0004524C MOV  X0, X23
LOAD:00045250 STR  W1, [X20,#0x14]
LOAD:00045254 LDR  W1, [SP,#0x430+arg_8]
```

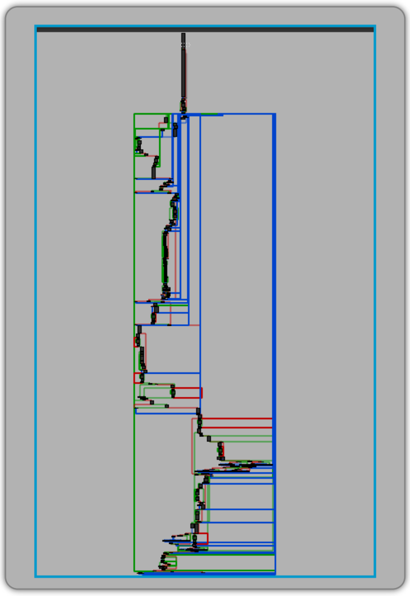


Figure 13 – Basic block of interest in the obfuscated function

The figure 14 is a section of the call trace generated by QBDI in which we can clearly see the calls to `prctl` and `setrlimit` to disable debugging and dump.

```
0x045200 prctl(SET_DUMPABLE)
0x045220 setrlimit(CORE)
0x045224 call getpid(): 21935
0x045240 prctl(SET_PTRACER, 21935)
 ...
0x045320 malloc(0x18): 0x74891eacc0
0x04532c malloc(0x40): 0x74891f6a40
0x045348 env->GetObjectClass(0x7fd0440cb8): 0x75
0x045424 env->GetMethodID('0x75', 'getPackageManager', '()Landroid/content/pm/PackageManager;')
0x0454d0 env->GetMethodID('0x75', 'getPackageName', '()Ljava/lang/String;'): 0x71cb6310
0x0454ec env->CallObjectMethod(0x7fd0440cb8, 0x75.getPackageManager, ... )
0x045504 env->GetObjectClass(0x85): 0x99
0x045604 env->GetMethodID('0x99', 'getPackageInfo', '(Ljava/lang/String;I)Landroid/content/pm/ ... )
0x045624 env->CallObjectMethod(0x7fd0440cb8, 0x75.getPackageName, ... )
0x045644 env->CallObjectMethod(0x85, 0x99.getPackageInfo, ... )
0x04565c env->GetObjectClass(0xb9): 0xc1
0x045714 env->GetFieldID(0xc1, 'signatures', '[Landroid/content/pm/Signature;'): 0x71b39c8c
0x04572c env->GetObjectField(0x0000b9, 0xc1.signatures): [Landroid.content.pm.Signature;
0x045740 env->GetObjectArrayElement( ... )
0x04575c env->GetObjectClass(0xe5): 0xf9
0x0457d8 env->GetMethodID('0xf9', 'toByteArray', '()[B'): 0x71b8d8e8
0x0457f0 env->CallObjectMethod(0xe5, 0xf9.toByteArray, ... )
0x045808 env->GetByteArrayElements( ... )
0x045824 env->GetArrayLength(0x101): 0x37b
```

Figure 14 – Call trace generated by QBDI

Since QBDI is a DBI and not a debugger, `ptrace` is not detected and since we are in the memory space of the application, we can arbitrarily dump any address.

4.2 Unpacking

The main purpose of the packer is to protect the original DEX files by encrypting them in the APK. When the application starts, it runs a routine that decrypts and loads the original DEX files. The encrypted DEX files — `classesX.dat` on the figure 15 — are embedded in the `assets/` directory along with the encrypted resources (`resources.dat`).

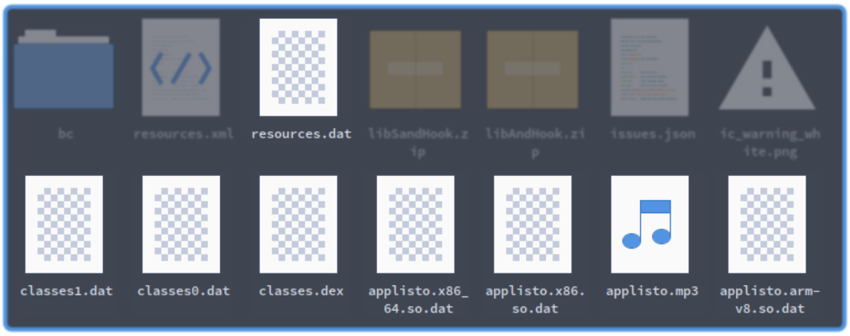


Figure 15 – `assets/` directory that contains encrypted DEX files

Using QBDI, we can generate a trace[16] of the unpacking routine to figure out how the packer uses the encrypted DEX files, and at which point they are likely to be decrypted in memory. Since QBDI is a DBI, not a debugger, the previous anti-debug were not triggered and we did not have to bypass the protections implemented in the library.

At some point, the uncompressed and decrypted DEX files are present in a memory buffer dynamically allocated with `mmap` (figure 16). As we can track dynamic memory allocations with QBDI (i.e. `mmap` and `malloc`), we can wait that the unpacking routine finishes and then inspect these memory buffers.

```
0x04a710 call 0x4a224 {
    0x04a264 call 0x48cec {
    }
    0x04a360 memcmp(AndroidManifest.xml, assets/classes1.dat, 0x13)
    0x04a360 memcmp(assets/applisto.mp3, assets/classes1.dat, 0x13)
    0x04a360 memcmp(assets/classes0.dat, assets/classes1.dat, 0x13)
    0x04a360 memcmp(assets/classes1.dat, assets/classes1.dat, 0x13)
}
0x04a7ec call 0x49f10 {
                        0xa501c => Size of classes1.dat
    0x049f5c mmap2(0x0, 0xa501c, 1, 1): 0x748275a000
}
0x043404 call 0x417c0 {
  0x041858 call 0x250e0 {
  }
  0x041884 call 0x41600 {
      0x041668 call 0x3d120 {
      }
      0x041704 call 0x3d120 {
      }
      0x041728 mmap2(0x0, 0x179b20, 3, 34): 0x747f6df000
      0x041754 call 0x201e4 {
          0x02027c call 0x57b18 {
              0x057a64 malloc(0x1bf0): 0x7471c58400
              0x057a80 call 0x57964 {
              }
          }
```

Figure 16 – Section of the trace in the unpacking routine

While this technique is well known and quite simple, it's still efficient on this packer and we managed to recover the full[17] original DEX files (figure 17).

```
$ ls
mmap-748275a000.dump  mmap-748fa31000.dump  mmap-7416a56000.dump  mmap-7416b6600
mmap-742ace6000.dump  mmap-74dece1000.dump  mmap-74de000000.dump  mmap-73a112a00
$ file ./mmap-748275a000.dump
./mmap-748275a000.dump: Dalvik dex file version 035
```

Figure 17 – Memory buffer that contains a plain DEX file

[16] A call trace is enough
[17] Some packers like Tencent's one remove parts of the DEX files so that a dump is not enough to recover the original code

4.3 Video game protection

Because Android video games are becoming more and more popular, they have to deal with similar threats as desktop games: bots, cheat, premium items fraud, ...

These attacks are usually performed by altering statically and/or dynamically the behavior of the application with tools such as:

- **Apktool**: repackaging
- **Frida**: Application-wide hooking
- **Xposed**: System-wide hooking
- **Lucky Patcher**: patching

By looking at a famous video game, we found several of these protections:

- Anti bot: Java layer
- Anti emulator: Java layer
- Anti repackaging: Java layer
- **Anti Frida: native obfuscated layer**

We focused the analysis on the way the application manages to detect Frida as it is located in a native library and protected by a commercial obfuscator.

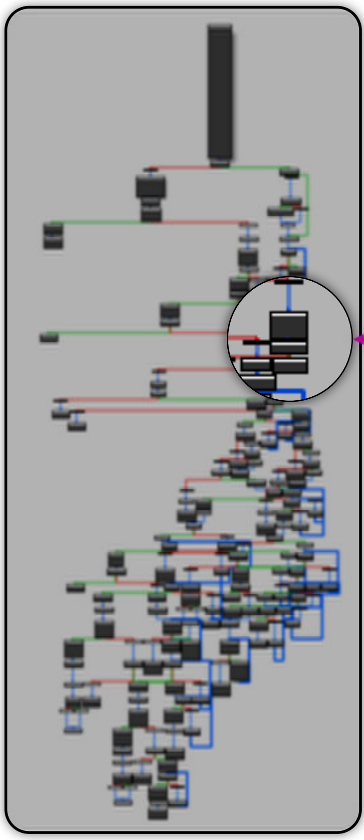
When the game starts with the Frida server running in background, we can notice that the application crashes with the backtrace exposed in the figure 18. From this backtrace, we can see that the crash comes from the `GameApp.createGameMain()` JNI function.

```
// adb logcat -s "*:F"
F libc    : Fatal signal 31 (SIGSYS), code 1 (SYS_SECCOMP) in tid 15317 (ll.XXX), pid 15317 (ll.XXX)
F DEBUG   : *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
F DEBUG   : Build fingerprint: 'google/taimen/taimen:9/PPR2.180905.005/4928864:user/release-keys'
F DEBUG   : Revision: 'rev_10'
F DEBUG   : ABI: 'arm'
F DEBUG   : pid: 15317, tid: 15317, name: ll.XXX  >>> com.XXX.YYY <<<
F DEBUG   : signal 31 (SIGSYS), code 1 (SYS_SECCOMP), fault addr --------
F DEBUG   : Cause: seccomp prevented call to disallowed arm system call -6047136
F DEBUG   :     r0  d29a6c90  r1  ff861b44  r2  ffa384c0  r3  ffa384c0
F DEBUG   :     r4  ffa384c0  r5  ffa384c0  r6  ffa384c0  r7  ffa3ba60
F DEBUG   :     r8  00000032  r9  ffa374c0  r10 ffa384c0  r11 d3104d1c
F DEBUG   :     ip  ffa3ba60  sp  ffa36518  lr  d2ba9445  pc  d29a6c94
F DEBUG   :
F DEBUG   : backtrace:
F DEBUG   :     #00 pc 00051c94  /data/app/com.XXX.YYY-XHpL30LP7HmGbCY2f8GIZw==/lib/arm/libg.so
F DEBUG   :     #01 pc 0050f82b  /data/app/com.XXX.YYY-XHpL30LP7HmGbCY2f8GIZw==/lib/arm/libg.so
F DEBUG   :     #02 pc 0050f82b  /data/app/com.XXX.YYY-XHpL30LP7HmGbCY2f8GIZw==/lib/arm/libg.so
F DEBUG   :     #03 pc 00021253  /data/app/com.XXX.YYY-XHpL30LP7HmGbCY2f8GIZw==/oat/arm/base.odex (com.XXX.YYY.GameApp.createGameMain+23
F DEBUG   :     #04 pc 0040d575  /system/lib/libart.so (art_quick_invoke_stub_internal+68)
F DEBUG   :     #05 pc 003e6c7b  /system/lib/libart.so (art_quick_invoke_static_stub+222)
```

Figure 18 – Backtrace when Frida server is running

By instrumenting this function with QBDI, we can observe that the `createGameMain` function spawns three threads (figure 19) that turned out to be involved in the detection routine [18]. After analysis, the first thread's routine tries to connect to Frida server by scanning all the ports periodically. If it manages to communicate with Frida server, it makes the application crash.

[18] Even though QBDI was not designed with built-in thread support, its design enables to deal with multi-threaded targets



```
.text:000A9C2C
.text:000A9C2C loc_A9C2C
.text:000A9C2C LDR      R0, =(unk_A79DB8 - 0×A9C34)
.text:000A9C2E LDR      R1, =(createGameMain_thread_1+1 - 0×A9C36)
.text:000A9C30 ADD      R0, PC ; unk_A79DB8 ; once_control
.text:000A9C32 ADD      R1, PC ; createGameMain_thread_1 ; init_routine
.text:000A9C34 BLX      pthread_once
.text:000A9C38 ADD.W    LR, SP, #0×7680+var_7658
.text:000A9C3C MOVS     R0, #4
.text:000A9C3E ADD.W    R2, LR, #0×6F80
.text:000A9C42 STR.W    R0, [R2,#0×1BC]
.text:000A9C46 B        loc_A9C1C
```

```
0×092bda delete(0×926e3900): /system/lib/libOpenSLES.so
0×092c64 delete(0×b9ff6d98): l_!9nddd
0×0a9c34 pthread_once(libg.so@0×a79db8, libg.so@0×1594e1)
0×0a9c68 pthread_once(libg.so@0×a76754, libg.so@0×3097cd)
0×0a9cb2 pthread_once(libg.so@0×a7382c, libg.so@0×4ffccd)
0×0a9cb8 .text!0×409f14 (#0) {
   ...
}
```

Figure 19 – Thread creation in `GameApp.createGameMain()`

The figure 20 shows the part of the trace that tries to connect to Frida's socket while the figure 21 shows the instructions associated with the `bind(27, 127.0.0.1, 41577)`.

```
0×2516dc .text!0×44c9bc (#484) {
0×2516dc .text!0×44c9bc (#485) {
0×2516dc .text!0×44c9bc (#486) {
0×2516dc .text!0×44c9bc (#487) {
0×251750 __errno()
0×251784 socket(IPV4, TCP, 0)
0×252eea __errno()
0×251c22 setsockopt(27, SOCKET, RCVTIMEO)
0×2518f6 bind(27, 127.0.0.0.1, 41577)
0×251980 __errno()
0×252dfa __errno()
0×2519be __errno()
0×251a0c syscall close()
0×24d854 free(0×92a57e00): @C ~/wlan0
0×24d854 free(0×92a57c40): z}`Vjdummy0
0×24d854 free(0×92a57a80): x Iz {lo
```

Figure 20 – Trace of the first thread

```
.text:002518B8 loc_2518B8 ; CODE XREF: .text:loc_251C74
.text:002518B8            ; .text:00252E9A
.text:002518B8 ADD.W      LR, SP, #8
.text:002518BC ADD.W      R0, LR, #0x1FA0
.text:002518C0 LDR.W      R3, [R0,#0x5A8]
.text:002518C4 LDR.W      R6, [R0,#0x584]
.text:002518C8 ADD.W      LR, SP, #8
.text:002518CC MOV.W      R4, #0x11A
.text:002518D0 ADD.W      R0, LR, #0x1FA0
.text:002518D4 MOV.W      LR, #0x10
.text:002518D8 MOV        R5, R0
.text:002518DA MOVS       R0, #0
.text:002518DC STR.W      R3, [R5,#0x778]
.text:002518E0 STR.W      R6, [R5,#0x77C]
.text:002518E4 STR.W      LR, [R5,#0x774]
.text:002518E8 STR.W      R0, [R5,#0x770]
.text:002518EC MOV        R0, R6
.text:002518EE MOV        R1, R3
.text:002518F0 MOV        R2, LR
.text:002518F2 MOV        R12, R7
.text:002518F4 MOV        R7, R4
.QBDI          Callback
.text:002518F6 SVC        0
.text:002518F8 MOV        R7, R12
.text:002518FA MOV        R3, R0
.text:002518FC STR.W      R3, [R5,#0x770]
.text:00251900 CMP        R3, #0
.text:00251902 BLT        loc_25195C
.text:00251904 B          loc_251932
```

bind() syscall

Figure 21 – Syscall `bind()`

As we can see in the figure 21, the code does not use the standard libc's function bind() but prefers to make a syscall. It makes sense since the function aims to detect Frida which could be used to hook the libc's bind function and to return a fake value. Nevertheless, this code is a good example to show how QBDI is working at instruction level.

Finally, we can **persistently** patch the library with LIEF to replace the original syscall instruction with a `"mov r0, #-1"` [19].

4.4 Root detection in a mobile device management application

A Mobile Device Management (MDM) is a software-based solution that provides features for companies to manage a large number of devices and to apply company's policies.

Usually, these solutions do not allow rooted devices as it would increase the attack surface. Thus, they are likely to have a strong and reliable mechanism to detect such violations.

Trying to obfuscate a root detection routine by a third-party application[20] is not easy since detecting the device's root status involves communicating with the system and therefore, calling library's API. While obfuscators can statically encode strings or data, the parameters going through external functions[21] need to be decoded and not obfuscated. One can think about the `open()` function: its first parameter needs to be the clear path to file to open. Not an encoded buffer. Therefore in this kind of analysis, if we are only interested in understanding how the application detects the device root status, it's mostly a matter of how

[19] `https://gist.github.com/romainthomas/f25b0377d8f0f37601c9a223e2105f32`
[20] that is not own by Google neither the device constructor
[21] Those for instance that are imported from `libc.so`

to trace external calls. We can hook all the functions that are likely to be called or we can trace the code with QBDI and observe the external calls.

During our tests of QBDI, we identified a MDM solution that implements a root detection in a JNI function obfuscated with the same obfuscator as the previous example. The figure 22 represents the CFG of this function.

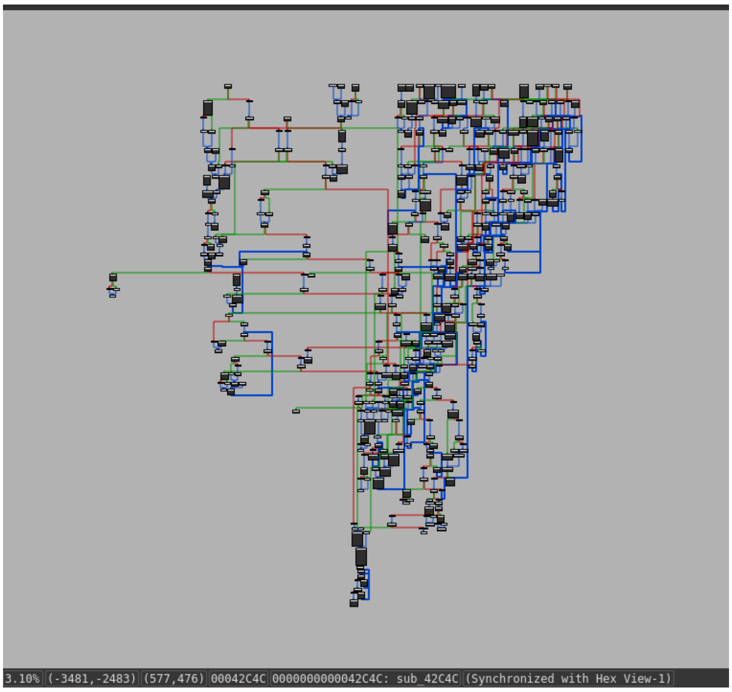


Figure 22 – `getDeviceState()` CFG

Similarly to the previous use cases, we based our analysis on a dynamic trace generated by QBDI which led to the trace in figure 23.

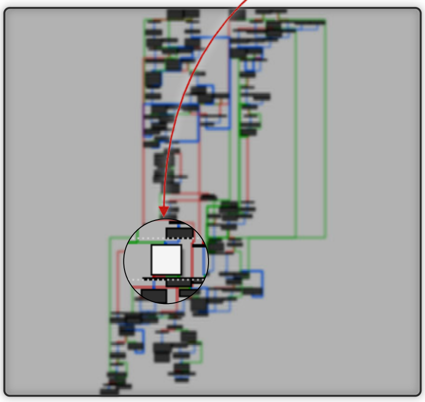
```
0x069c60 strlen(' |;} |q#p";z|qry')
0x0695f4 strlen(' |;} |q#p";z|qry')
0x06980c strlen('ro.product.model')
0x069ea4 memcpy(0x72ccbea479, 0x72ccbea140, 0x10): ro.product.model
0x069c60 strlen(' ||"')
0x0695f4 strlen(' ||"')
0x06980c strlen('root')
0x069ea4 memcpy(0x72ccbea479, 0x72ccbea150, 0x4): root
0x069c60 strlen('y|-:y_-◁'|"rz-*-t r}-:r-G1-:r-/k:h :jh%:j&/')
0x0695f4 strlen('y|-:y_-◁'|"rz-*-t r}-:r-G1-:r-/k:h :jh%:j&/')
0x06980c strlen('ls -lR /system | grep -e :$ -e "^[r-][w-]x"')
0x0b80f0 malloc(0x30): 0x72d08ed620
0x069ea4 memcpy(0x72d08ed620, 0x72ccbea130, 0x2c): ls -lR /system | grep -e :$ -e "^[r-][w-]x"
0x069c60 strlen('`#}r `b')
0x0695f4 strlen('`#}r `b')
0x06980c strlen('SuperSU')
0x069ea4 memcpy(0x72ccbea479, 0x72ccbea150, 0x7): SuperSU
0x069c60 strlen('zn"pur|')
0x0695f4 strlen('zn"pur|')
0x06980c strlen('matches')
0x069ea4 memcpy(0x72ccbea479, 0x72ccbea150, 0x7): matches
0x069c60 strlen('} |pzrz')
0x0695f4 strlen('} |pzrz')
0x06980c strlen('procmem')
0x069ea4 memcpy(0x72ccbea479, 0x72ccbea150, 0x7): procmem
```

Figure 23 – Call trace that suggests string decoding

In this trace, we can notice a pattern that looks like a string decoding routine.

At address `0x69c60`, we can see that the encoded string going through `strlen()` has the same length as the clear string used at address `0x06980c`.

By statically looking at the basic block that covers the first address (figure 24), we can identify the algorithm used to decode the strings.



```
.text:0000000000069C58 loc_69C58
.text:0000000000069C58 STUR      X8, [X29,#-0×D0]
.text:0000000000069C5C LDR       X0, [X23,X8,LSL#3]
.text:0000000000069C60 BL        .strlen
.text:0000000000069C64 MOV       X8, SP
.text:0000000000069C68 STUR      X8, [X29,#-0×E0]
.text:0000000000069C6C ADD       X8, X0, #0×10
.text:0000000000069C70 AND       X8, X8, #0×FFFFFFFFFFFFFFF0
.text:0000000000069C74 MOV       X9, SP
.text:0000000000069C78 SUB       X8, X9, X8
.text:0000000000069C7C MOV       SP, X8
.text:0000000000069C80 STR       X8, [X19,#0×178]
.text:0000000000069C84 LDR       X8, [X19,#0×30]
.text:0000000000069C88 LDR       X9, [X8]
.text:0000000000069C8C LDUR      X8, [X29,#-0×C0]
.text:0000000000069C90 LDR       X8, [X8]
.text:0000000000069C94 SUB       X9, X9, X8
.text:0000000000069C98 ASR       X9, X9, #3
.text:0000000000069C9C STR       X9, [X19,#0×F8]
.text:0000000000069CA0 LDUR      X10, [X29,#-0×D0]
.text:0000000000069CA4 STR       X10, [X19,#0×F0]
.text:0000000000069CA8 BIC       X11, X9, X10
.text:0000000000069CAC ORN       X12, X9, X10
.text:0000000000069CB0 SUB       X9, X10, X9
.text:0000000000069CB4 AND       X9, X12, X9
.text:0000000000069CB8 ORR       X9, X11, X9
.text:0000000000069CBC STR       X9, [X19,#0×E8]
.text:0000000000069CC0 LSR       X9, X9, #0×3F
.text:0000000000069CC4 CBNZ      X9, loc_699F4
```

Figure 24 – Basic block involved in the string decoding routine

One of the clear strings is associated with a command: `ls -lR /system | grep -e :$ -e "-[r-][w-]x"`. that is used later in the trace by `popen()` (figure 25).

```
0×03febc call 0×6d4ac {
    0×06d5b4 memset(0×72ccbe9ab0, 0, 0×400)
    0×06d5c0 pthread_mutex_lock()
    0×06d964 popen('ls -lR /system | grep -e :$ -e "^[r-][w-]x", ... )
    0×06d98c fileno()
    0×06d9a8 poll()
    0×06d70c fgets( ... ): /system:
    0×06d868 strlen('/system:')
    0×06d52c memcpy(0×72ccbe9a99, 0×72ccbe9ab0, 0×9): /system:
    0×06d6a0 call 0×78194 {
        0×0b80f0 malloc(0×18): 0×72d015e500
    }
    0×06d614 fgets( ... ): /system/app:
    0×06d868 strlen('/system/app:')
    0×06d52c memcpy(0×72ccbe9a99, 0×72ccbe9ab0, 0×d): /system/app:
    0×06d6a0 call 0×78194 {
        0×0b80f0 malloc(0×30): 0×72d01eb800
        0×0782a0 free(0×72d015e500):
    }
    0×06d614 fgets( ... ): /system/app/BasicDreams:
```

Figure 25 – Use of the decoded string

The capacity to identify where and when the data are used could be decisive if we aim to patch the library to disable some of these protections. One could also craft a custom output in the instrumentation callback that would hide the distinctive features of a rooted device.

Going further in the trace, the function `getDeviceState()` opens the `/proc/net/unix` file to detect if some entries are associated with Magisk. In the figure 26 we can see the sequence of functions that check if Magisk is present.

```
0x056fac fopen('/proc/net/unix', ... )
0x052e78 strcpy()
0x052e84 call 0x5f904 {
  0x05fad0 strlen('MAGISK')
  0x05faec fseek(72eb4140b0, 0, 2)
  0x05faf4 ftell(493573193904): 0x0
  0x05fb00 malloc(0x0): 0x72d0f4eac0
  0x05fba4 strlen('MAGISK')
  0x05fbc4 fread(0x72ccbea0b8, 0x1, 0x400): Num       RefCount Protocol Flags    Type St Inode Path0000000000000000: 00000002 00000000 000100 ...
  0x05fa24 memmem(Num       RefCount Protocol Flags    Type St Inode Path0000000000000000: 00000002 00000000 000100 ... , MAGISK)
  0x05fb98 fseek(72eb4140b0, -70, 1)
  0x05fab0 fread(0x72ccbea0b8, 0x1, 0x400): dev/socket/qmux_radio/uim_remote_client_socket0000000000000000: 00000002 00000000 00010000 0001  ...
  0x05fa24 memmem(dev/socket/qmux_radio/uim_remote_client_socket0000000000000000: 00000002 00000000 00010000 0001  ... , MAGISK)
  0x05fb98 fseek(72eb4140b0, -70, 1)
  0x05fab0 fread(0x72ccbea0b8, 0x1, 0x400): 00010000 0001 01 18810 /dev/socket/logd0000000000000000: 00000002 00000000 00010000 0005 01 18815 ...
  0x05fa24 memmem(00010000 0001 01 18810 /dev/socket/logd0000000000000000: 00000002 00000000 00010000 0005 01 18815 ... , MAGISK)
  0x05fb98 fseek(72eb4140b0, -70, 1)
  0x05fab0 fread(0x72ccbea0b8, 0x1, 0x400): et/tombstoned_intercept0000000000000000: 00000002 00000000 00010000 0005 01 23990 /dev/socket/tom ...
  0x05fa24 memmem(et/tombstoned_intercept0000000000000000: 00000002 00000000 00010000 0005 01 23990 /dev/socket/tom ... , MAGISK)
  0x05fb98 fseek(72eb4140b0, -70, 1)
  0x05fab0 fread(0x72ccbea0b8, 0x1, 0x400): 00 00010000 0001 01 30193 /dev/socket/qmux_radio/qcril_radio_config0000000000000000: 00000002 00 ...
  0x05fa24 memmem(00 00010000 0001 01 30193 /dev/socket/qmux_radio/qcril_radio_config0000000000000000: 00000002 00 ... , MAGISK)
  ...
```

Figure 26 – Magisk detection based on `/proc/net/unix`

Last but not least, the MDM library seems to be written in `C++` which is sometimes more challenging to reverse than C but on the other hand, language's properties may help. Let's consider the code in figure 27

```cpp
#include <string>

void decode(char& c) {
  c ^= 0x33;
}

int check_root(const std::string& input) {
  std::string encoded = input;
  for (char& c : encoded) {
    decode(c);
  }
  // [IMPLICIT CALL] operator delete(void*); ⟵ decoded
  return 0;
}
```

Figure 27 – C++ code with implicit destructor

In the function `check_root()` there is a `std::string` object allocated on the stack but whose the **internal buffer is dynamically allocated**[22].

The standard requires that when stack object goes out of its scope — in this case at the end of the function — its destructor is automatically invoked. In this case automatically means generated by the compiler.

Therefore, there is an implicit `operator delete()` at the end of the function that releases the internal buffer of `std::string`. At the assembly level, it behaves as an external call that can be caught by QBDI's ExecBroker.

At the end of the trace a lot of memory buffers are freed [23] and by inspecting these buffers we can have a good overview of the different root checks performed by the MDM solution. The figure 28 shows some parts of these buffers.

[22] For *small* strings this not true because of some optimizations

[23] `free()` and `operator delete()` have the same prototype and behavior therefore they are processed in a same *free* callback

```
checkOTACerts        → /data/app/ … /lib/arm64/libcoredevice.so@0×409b4
getDeviceState       → /data/app/ … /lib/arm64/libcoredevice.so@0×42b9c
getValues            → /data/app/ … /lib/arm64/libcoredevice.so@0×4111c
start                → /data/app/ … /lib/arm64/libcoredevice.so@0×67ca4
getValues2           → /data/app/ … /lib/arm64/libcoredevice.so@0×6e9d4
getString            → /data/app/ … /lib/arm64/libcoredevice.so@0×40714
getDeviceSalt        → /data/app/ … /lib/arm64/libcoredevice.so@0×6dbb4
getSeedValue         → /data/app/ … /lib/arm64/libcoredevice.so@0×6d9fc
getSeedValueV2       → /data/app/ … /lib/arm64/libcoredevice.so@0×40420
getRandomValueNative → /data/app/ … /lib/arm64/libcoredevice.so@0×6b7a0
isAppAllowed         → /data/app/ … /lib/arm64/libcoredevice.so@0×5f7dc
lookForMagiskV16     → /data/app/ … /lib/arm64/libcoredevice.so@0×68728
-----> Exit from JNI_OnLoad <-----
-----> Enter in getDeviceState <-----
 …
0×047230 free(0×72d09dd700): /system/bin/app_process64_xposedxrr
0×047230 free(0×72d09dd580): /system/bin/app_process32_xposedxrr
0×047230 free(0×72d0213b00): /system/bin/app_process64r
0×047230 free(0×72d0213ae0): /system/bin/app_process32r
0×047230 free(0×72d0213aa0): /system/bin/app_processipr
0×047230 free(0×72d019b4e0): grep --binary-files=text "Xposed" /system/bin/app_process64_xposedr8zr
0×047230 free(0×72d019b350): grep --binary-files=text "Xposed" /system/bin/app_process32_xposedrxr
0×047230 free(0×72d0160080): grep --binary-files=text "Xposed" /system/bin/app_process
0×047230 free(0×72d08cb810): /system/framework/XposedBridge.jarr(=
0×047230 free(0×72d0213a80): /etc/security/otacerts.zipr
0×047230 free(0×72d0213a00): com.ramdroid.appquarantiner
0×047230 free(0×72d08cb780): com.zachspong.temprootremovIsRoot.jb%
0×047230 free(0×72d08cb630): /system/usr/we-need-root/su-backupr!
0×047230 free(0×72d02139c0): com.koushikdutta.superuserr
0×047230 free(0×72d02139a0): com.thirdparty.superuserr
0×047230 free(0×72d02138a0): com.noshufou.android.su@r
0×047230 free(0×72d02138e0): /system/app/Superuser.apkr
0×04eb7c free(0×72d018a800): test-keys!8!r!8!r/data/system/bin/su/system/xbin/su'-c ls'sh idid(root)busybox df!9!r(eu.chainfir …
0×048840 free(0×72cf5ef300): /data//system/system/bin/system/sbin/system/xbin/vendor/bin/sys/sbin/etc/proc/devr
0×0594b0 free(0×72d08ed620): ls -lR /system | grep -e :$ -e "^[r-][w-]x"
0×0594b0 free(0×72d08ed770): ls -laR /system | grep [r-][w-]s[-r' ']QWK
0×0511f4 free(0×72d01b8000): /system/bin/sh'-c'1'prnetcfgpingrun-as /system/bin/grepdiag_mdlogls -l logamz_groupsuperSCH-I545p …
0×04b38c free(0×72d08b4120): getpropro.secure0
0×0475e0 free(0×72d03c1800): `r≠!r1%Ar1#Ar≠!r!package:com.nextdoorA=r1(`ArA=r1(Ar!r1'0ArA0r≠!r1"Ar1'PAr! >!r1$0ArA2@r1!Ar1. …
← Done
-----> Exit from getDeviceState <-----
```

Figure 28 – `std::string` buffers deallocated with `operator delete(void*)`

4.5 Legu Packer

Some parts of the reverse engineering of Legu[24][25] were done with Frida/QBDI. First, we started by getting an overview of the main library (`libshell-super.2019.so`) with a call trace starting from `JNI_OnLoad()`. Then we identified a global structure which were involved in the packer configuration (e.g. number of DEX files packed, Android OS version, Android runtime version, …). To figure out the meaning of the structure's fields, we first identified the library's instruction that allocates the structure through `calloc()` and we put a **single instruction callback** to get the structure size and its allocated address. Then, using QBDI memory callbacks we tracked all the memory reads and writes within this allocated buffer. Finally, thanks to the memory trace we managed to resolve most of the structure's fields statically. [26]

By going through these different levels — from call trace to a memory and instruction trace — we successfully managed to figure out the packer's logic.

A video that shows some parts of the analysis with Frida/QBDI is available here: `https://www.romainthomas.fr/publication/20-bh-asia-dbi/#demo2`

5. Conclusion

Even though code obfuscation can be a hassle for reverse engineers, it forces analysts to develop new techniques and new tools to handle such protections. Through this paper and the associated presentation, we aimed to present a set of DBI primitives which enables to extract program invariants[27] that can be difficult to protect even under a ton of obfuscation layers.

The assessments performed on these different applications also showed that the kind of obfuscation to protect the asset and the category of asset highly depend on each other. For instance, control-flow flattening does not really matter if we can dynamically trace the code. Similarly, the root detection in the MDM solution, is protected with classical code and data obfuscation but there is not protection against dynamic instrumentation.

[24] `https://blog.quarkslab.com/a-glimpse-into-tencents-legu-packer.html`
[25] `https://github.com/quarkslab/legu_unpacker_2019`
[26] Some parts of the library are obfuscated but a static analysis is still doable at basic block level
[27] One can think about a syscall in the original code: whatever the obfuscation/protection used, the syscall will be executed

A lot of thanks to Charles Hubain and Cédric Tessier who created and introduced the main concepts of QBDI as well as the initial support for x86/x86-64 and ARMv6. Thank you also to Nicolas Surbayrole, one of the principal developer and maintainer of QBDI. Finally, thanks to Tom Czayka, Laurent Laubin, Nicolas Surbayrole, and Damien Aumaitre for their feedbacks.

References

[1] Breaking Spotify DRM with PANDA, Recon 14, B. Dolan-Gavitt & al

[2] DRM obfuscation versus auxiliary attacks, Recon 14, C. Mougey & F. Gabriel

[3] API Deobfuscator Resolving Obfuscated API Functions In Modern Packers, BH15-USA, S. Choi

[4] Differential Computation Analysis:Hiding your White-Box Designs is Not Enough, SSTIC 16, C. Hubain, P. Teuwen

[5] Symbolic Deobfuscation: From Virtualized Code Back to the Original, Dimva 18, J. Salwan, S. Bardin & M. Potet

[6] Breaking Mobile Userland, NowSecure 19, G. Rocca

[7] Automation Techniques in C++ Reverse Engineering, Recon 19, R. Rolles

[8] Automation Techniques in C++ Reverse Engineering, Recon 19, R. Rolles

[9] https://qbdi.quarkslab.com/, C. Hubain, C. Tessier

[10] https://github.com/lief-project/LIEF, R. Thomas.

[11] https://blog.quarkslab.com/exploring-execution-trace-analysis.html

[12] https://gitlab.com/eshard/scared

[13] https://github.com/SideChannelMarvels/Daredevil