Endoscope: Unpacking Android Apps with VM-based Obfuscation

Speaker: Fan Wu

Contributor: Xuankai Zhang

About us

- Fan Wu: Expert Security Engineer at Meituan; previously with Alibaba Cloud.
  Speaker of multiple security conferences (BCS, CIS, etc.)

- Xuankai Zhang: Graduate student of ShangHai Jiaotong University and intern at Meituan.
  His area of interest is software reverse engineering and code analysis.

Agenda

- Intro & Background Story
- The Rhino Bytecode Case and its Reversing
- More General Scenario of VM-Packed Programs
- Unpacking under General Scenario
- Insights & Conclusion

VM-Based Obfuscation

Compiling code into bytecode comprising of a specific set of custom instruction, and running it on a custom-built Virtual Machine.

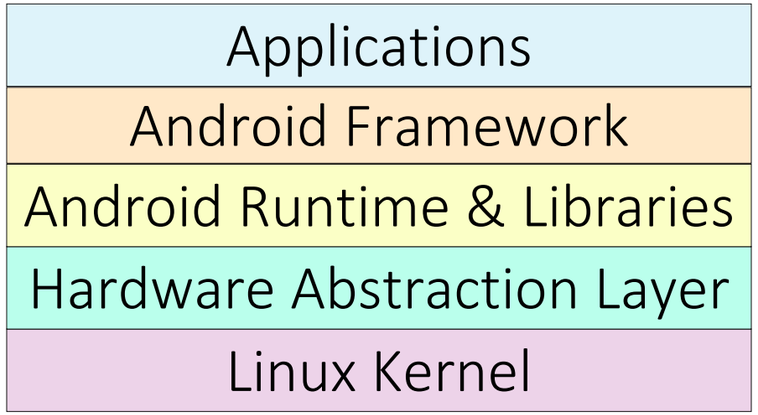
The tools used to carry out the obfuscation is referred to as a packer.

VM-Based obfuscation has long been used for benign and evil purposes

- Anti-plagiarism
- Intellectual property protection
- Hiding malicious payload

VM-Based Obfuscation on Android

- Recent years has seen increasing numbers of VM-protected Android apps
- Android has a multiple-layer architecture. Cross-layer invocations, especially those through Java Native Interface(JNI) are commonly used by Android VM-based packers, while those for PC programs have no such characteristic.



Background Story

- Months ago, when analyzing an Android malware, we found that it executes js bytecode on Mozilla Rhino engine, with the bytecode's source deliberately removed
- We studied the Rhino engine and managed to reverse and recover some of the malware's semantics

Background Story

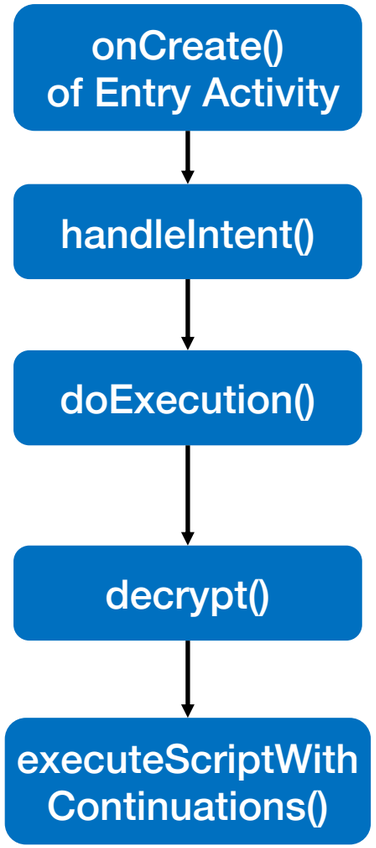
- Later, we encountered other in-the-wild malwares that are obfuscated with different virtualization techniques
- The exact method we used to reverse previous malware cannot be reused in these cases
- So we figured out a method that is more general

Custom Virtual Machine

Agenda

- Intro & Background Story
- The Rhino Bytecode Case and its Reversing
- More General Scenario of VM-Packed Programs
- Unpacking under General Scenario
- Insights & Conclusion

The Rhino Bytecode Malware Case



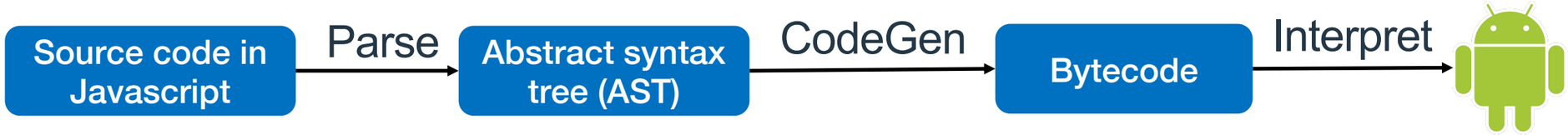

```
class InterpretedFunction {
    InterpreterData idata;
    Interpreter.interpret(this);
    ...
}
class InterpreterData {
    byte[] itsICode;
    String[] itsStringTable;
    BigInteger[] itsBigIntTable;
    String encodedSource;
    ...
}
```

```
public final class Interpreter{
    private static Object interpretLoop(...) {
        Object[] stack = frame.stack;
        int op = iCode[frame.pc++];
        switch (op) {
            case Icode_GENERATOR:
                ...
            case Token.YIELD:
                ...
        }
    }
}
```

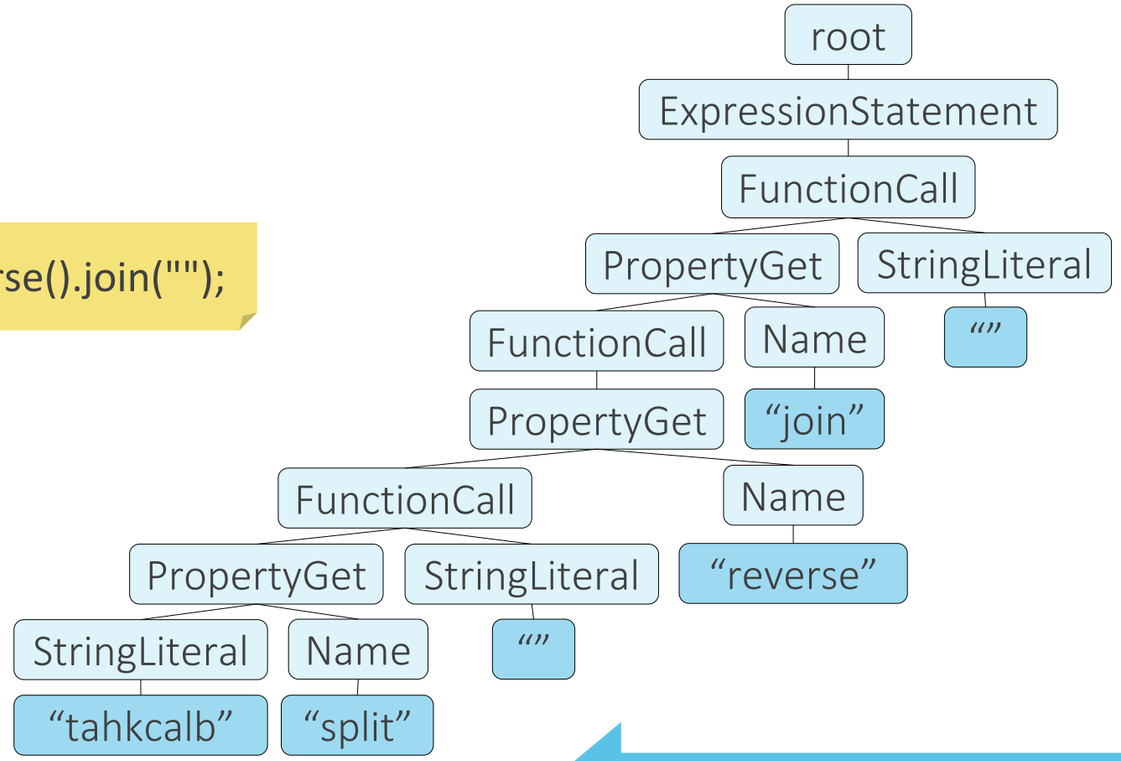
- This InterpretedFunction object only stores Rhino bytecode, not the source.
- Each byte of bytecode is interpreted in a large switch-case statement.

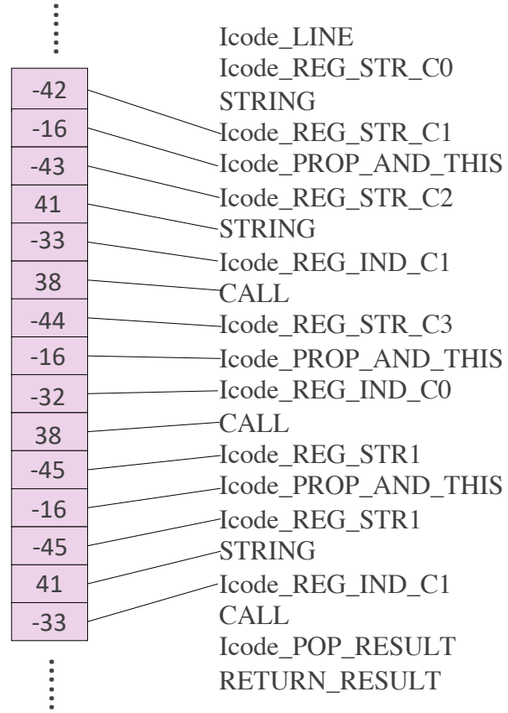
- Unlike the situation of Dalvik bytecode, there is no existing tool to translate Rhino bytecode back to source. And the encodedSource field of InterpreterData object is set to empty string.

Generation of Bytecode and its Reverse



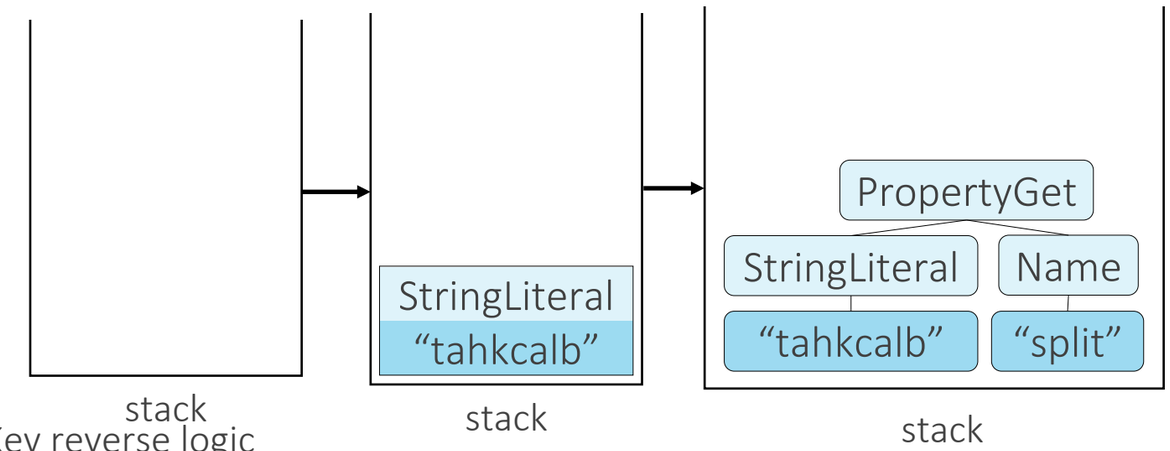

```
"tahkcalb".split("").reverse().join("");
```

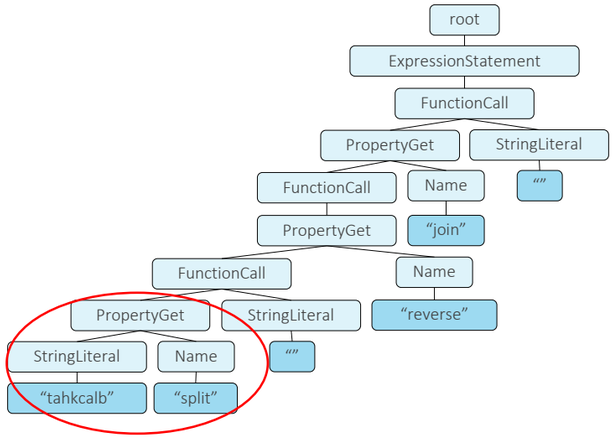




Reconstruct AST and Source

-42	Icode_LINE
-16	Icode_REG_STR_C0
-43	**STRING**
-42	Icode_REG_STR_C1
-16	**Icode_PROP_AND_THIS**
-43	Icode_REG_STR_C2
41	STRING
-33	Icode_REG_IND_C1
38	CALL
-44	Icode_REG_STR_C3
-16	Icode_PROP_AND_THIS
-32	Icode_REG_IND_C0
38	CALL
-45	Icode_REG_STR1
-16	Icode_PROP_AND_THIS
-45	Icode_REG_STR1
41	STRING
-33	Icode_REG_IND_C1
-42	CALL
-16	Icode_POP_RESULT
-43	RETURN_RESULT





Key reverse logic

```
Token.STRING -> {
    val sl = StringLiteral(0)
    sl.value=stringReg
    sl.quoteCharacter="'"
    stack.push(sl)
}
```

itsStringTable

0	”tahkcalb”
1	”split”
2	””

Rhino provides a toSource() method that transform an AST back into source code recursively ☺

```
Icode.Icode_PROP_AND_THIS -> {
    val func = FunctionCall()
    func.target = PropertyGet(stack.poll(), Name(0, stringReg))
    stack.push(func)
}
```

```
"tahkcalb".split("").reverse().join("");
```

Agenda

- Intro & Background Story
- The Rhino Bytecode Case and its Reversing
- More General Scenario of VM-Packed Programs
- Unpacking under General Scenario
- Insights & Conclusion

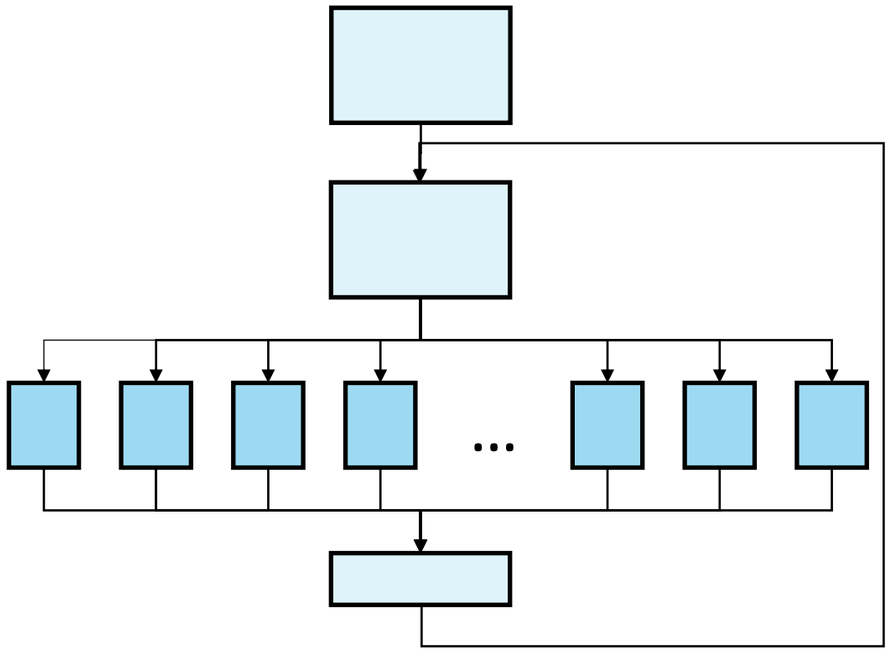
A more General VM-Packer Scenario

In-the-wild Samples protected by VM-based obfuscators on Android usually

- Close source
- Implement virtual machine in a native library (JNI)
- Adopt app-specific Randomization

Difficulty levels up!

Key Components: VM Entry / Exit



VM Entry: Switch to virtual context

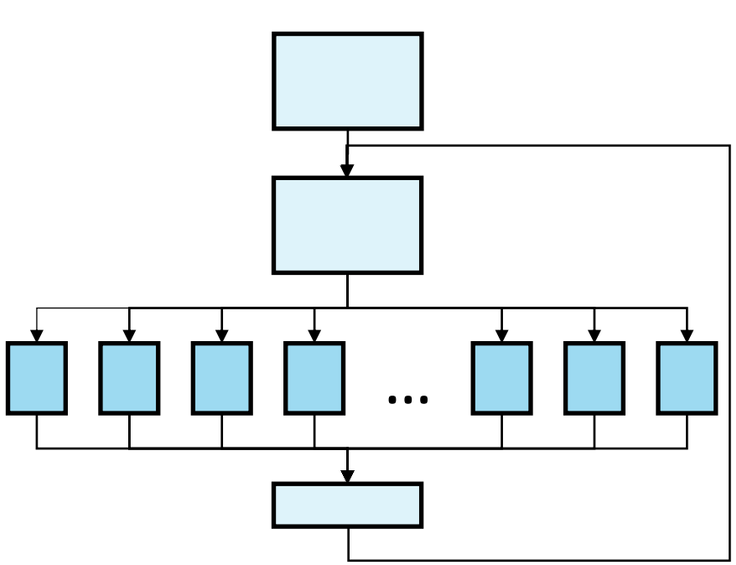
Copy registers, etc

Dispatcher

Handlers

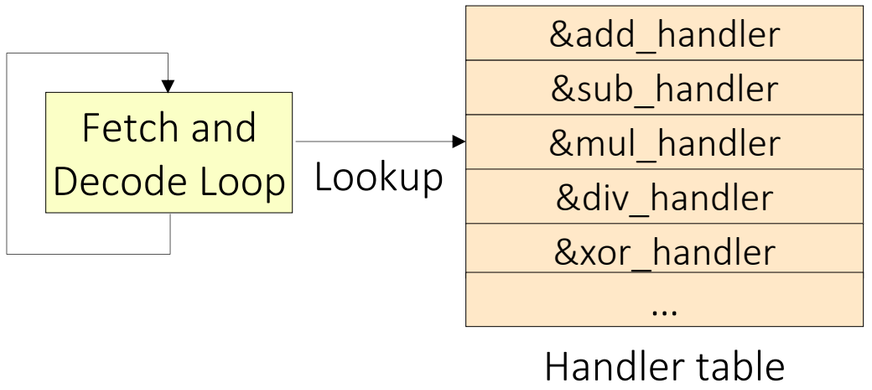
Loop / VM Exit: Switch back to original context

Key Components: Dispatcher & Handlers



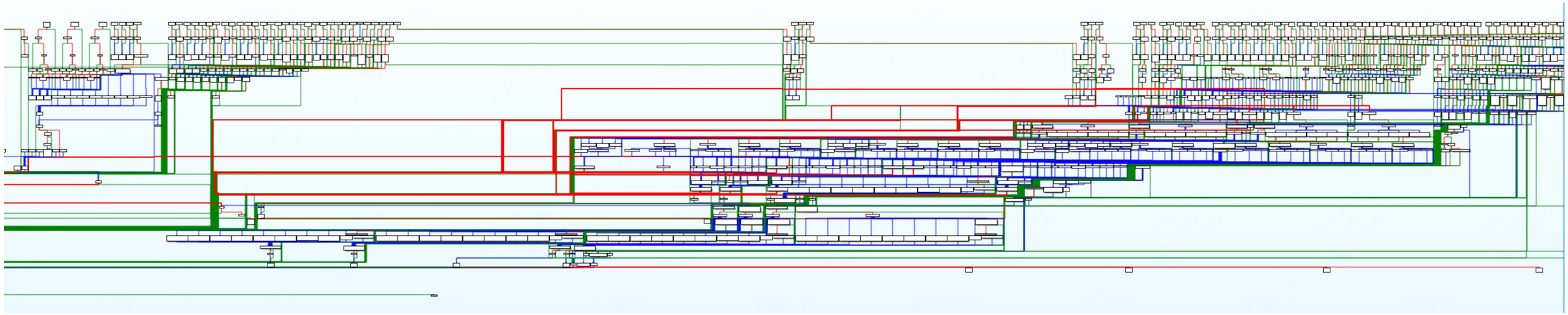
Dispatcher loop:

- Fetch and decode a virtual instruction
- Look up in the handler table
- Invoke the handler



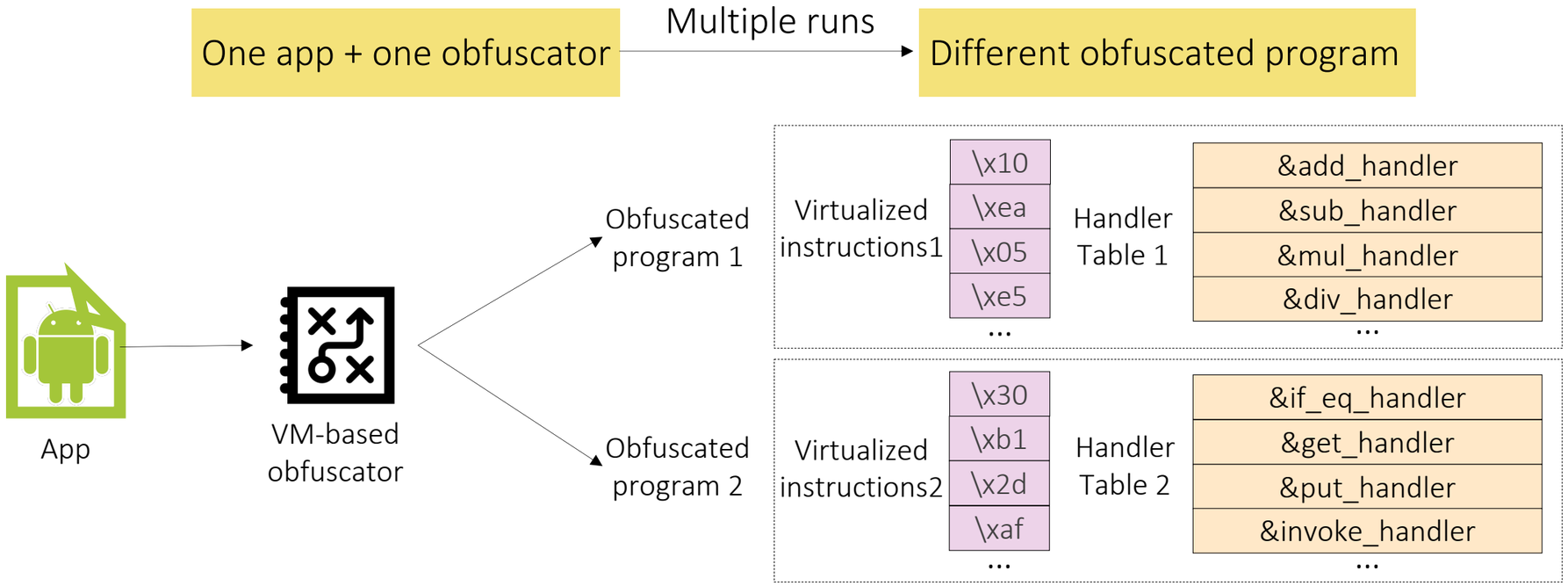
Challenges in Reversing

- We only have binaries for in-the-wild, not source code/original Dalvik bytecode.
- Lack of information about virtualized instructions and VM's mechanism
- Control flow is very complex, so static and dynamic analysis are both time-consuming



One more Challenge

- Many VM-protected programs use randomized, app-specific encryption parameters and order of handler pointers in handler table
- As a result, the result of manually analyzing a VM-packed program A, is not reusable to another program B which is packed with same packer



Agenda

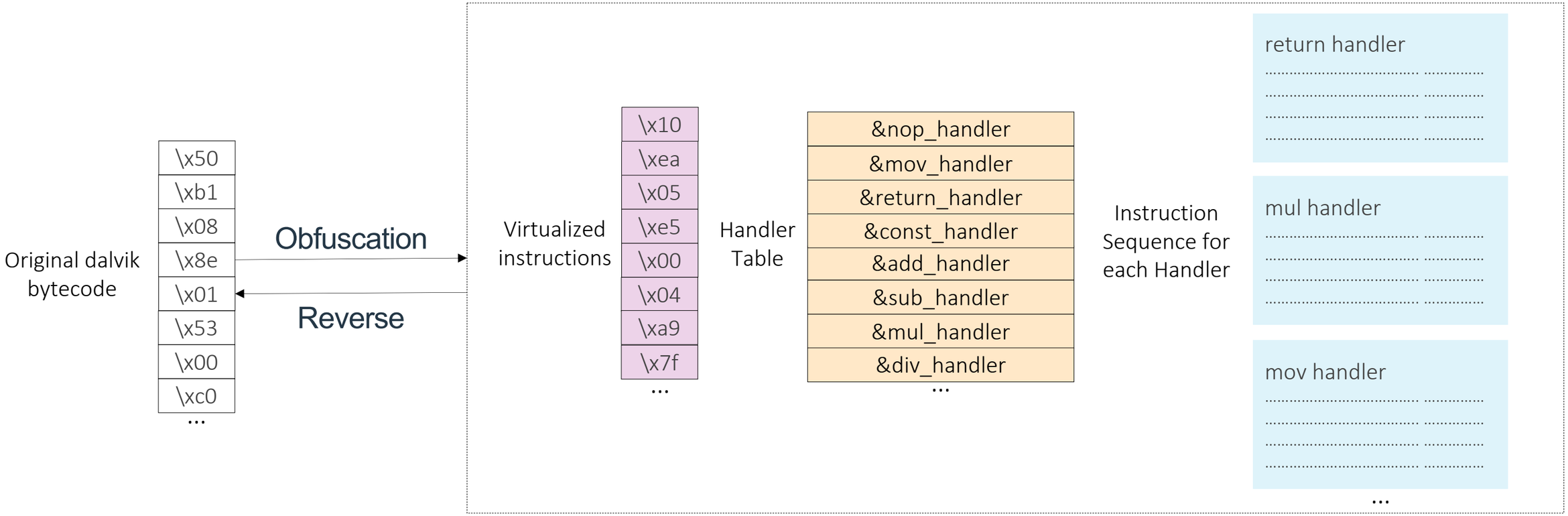
- Intro & Background Story
- The Rhino Bytecode Case and its Reversing
- More General Scenario of VM-Packed Programs
- Unpacking under General Scenario
- Insights & Conclusion

Assumptions and Prerequisites

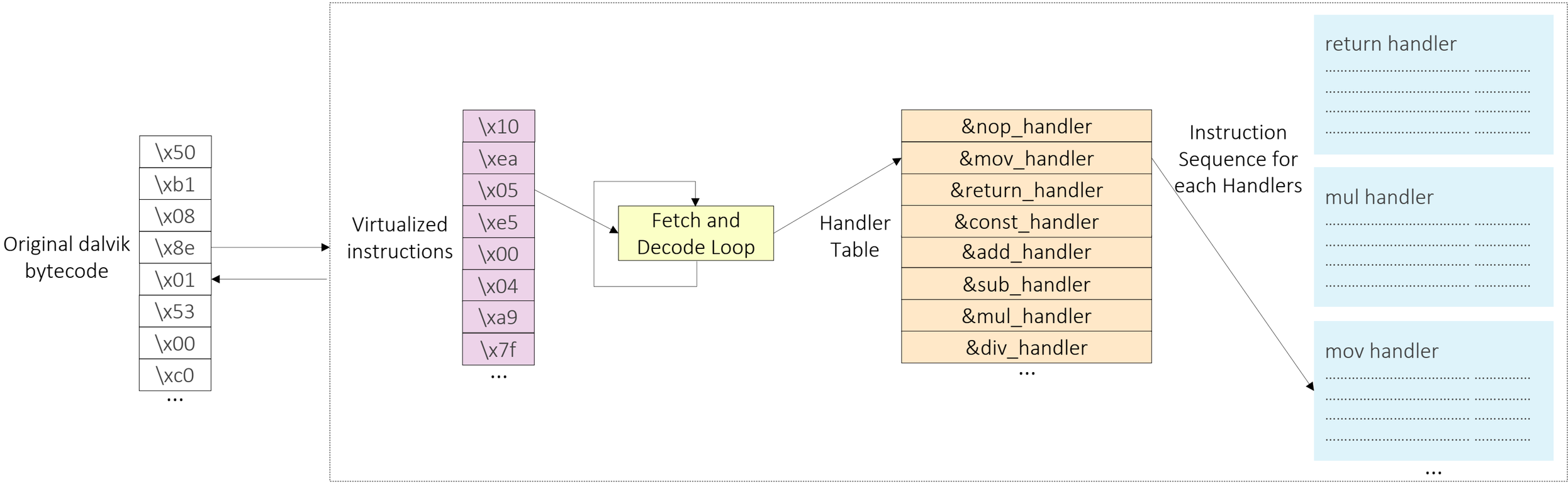
Our method of unpacking only works properly when the following assumptions are met

- The packer we are going to reverse is accessible and we can use it to obfuscate any custom app
- The obfuscator follow the typical pattern of “transform Dalvik bytecode into native functions”
- Provided that Dalvik bytecode can be easily transferred back to Java with existing tools, our goal under this scenario is to recover Dalvik bytecode from obfuscated program

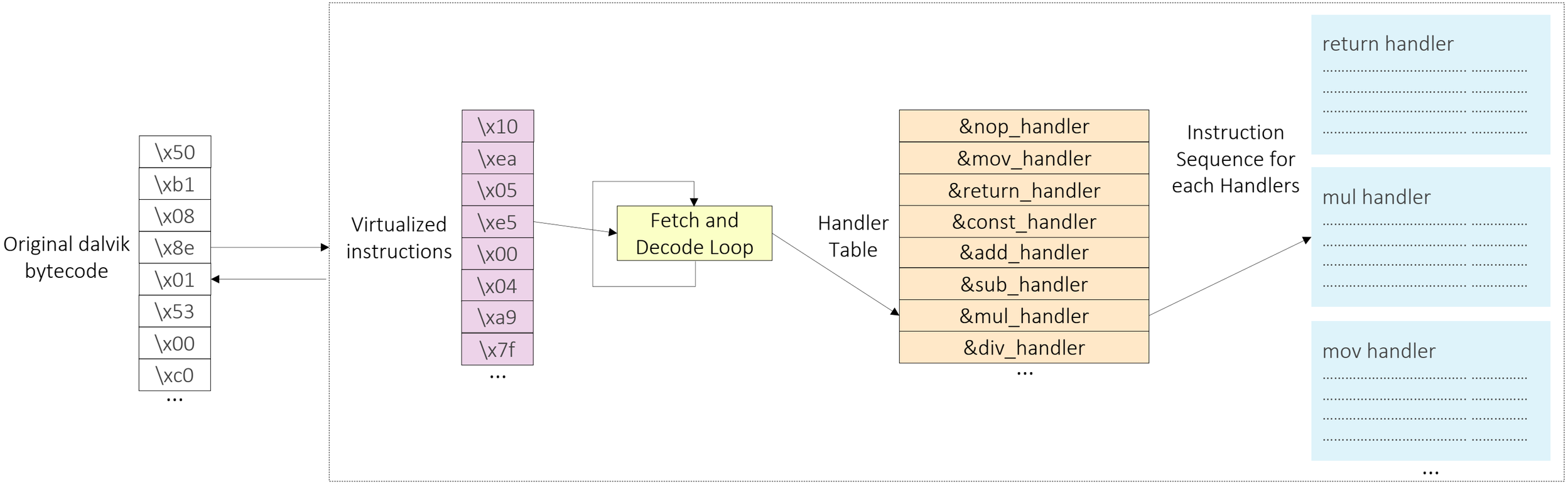
The Obfuscated Program



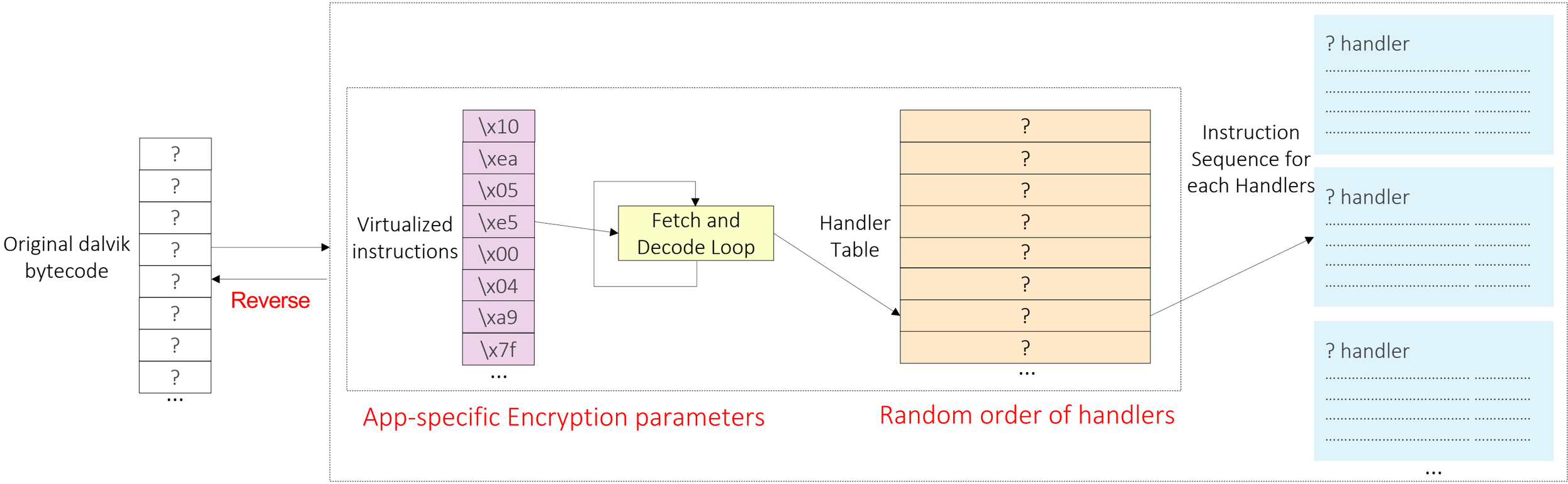
The Execution Process



The Execution Process Cont.



The Reverse Process



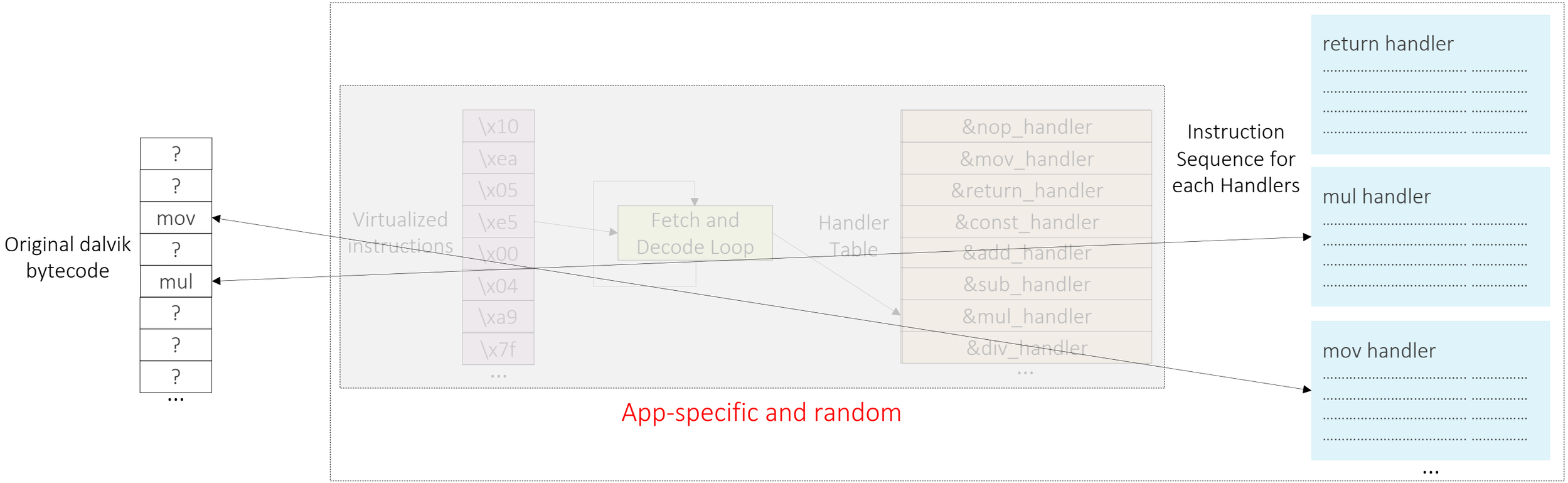
Intuition

- To keep the obfuscated program's semantics identical to that of original one, each handler of the VM is initially translated from a set of simple operations in original Dalvik bytecode
- Although the intermediates are app-specific, the relationship between the original Dalvik bytecode and the handler content is fixed (Diagram on next page)

- Also, the obfuscated functions generally pass parameters in the same way as original program.

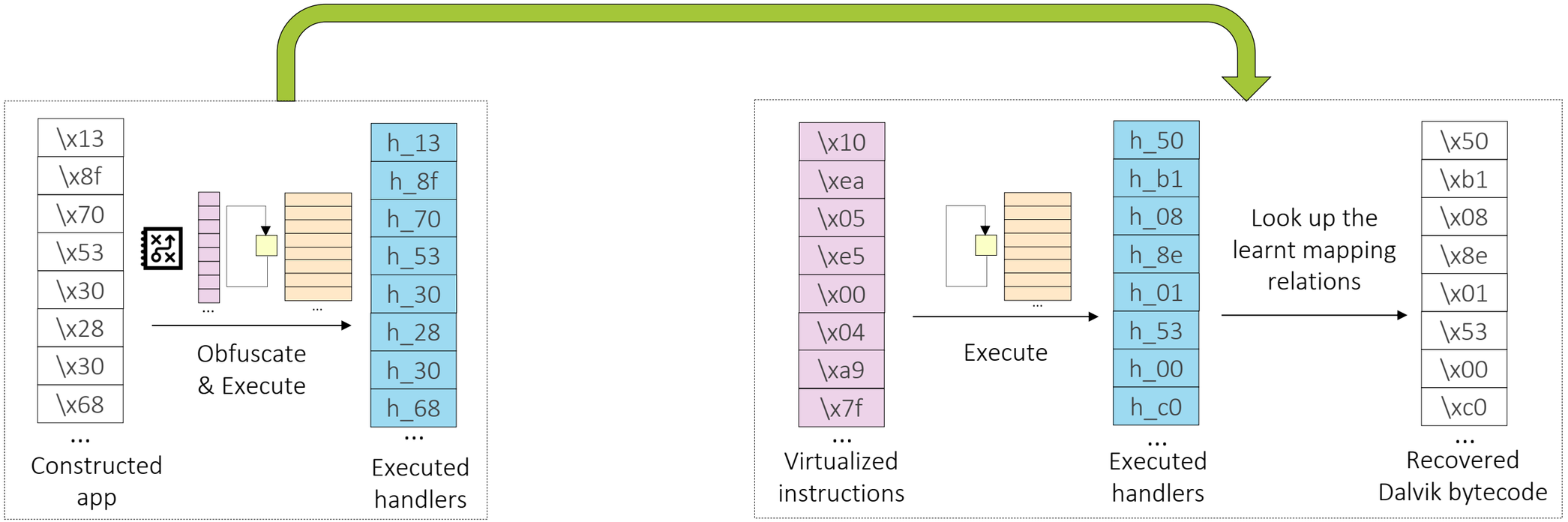
Intuition Cont.

- Whenever there is a mov instruction in the original Dalvik code, during execution of the obfuscated program the instruction sequence for mov handler must execute once, and vice versa.



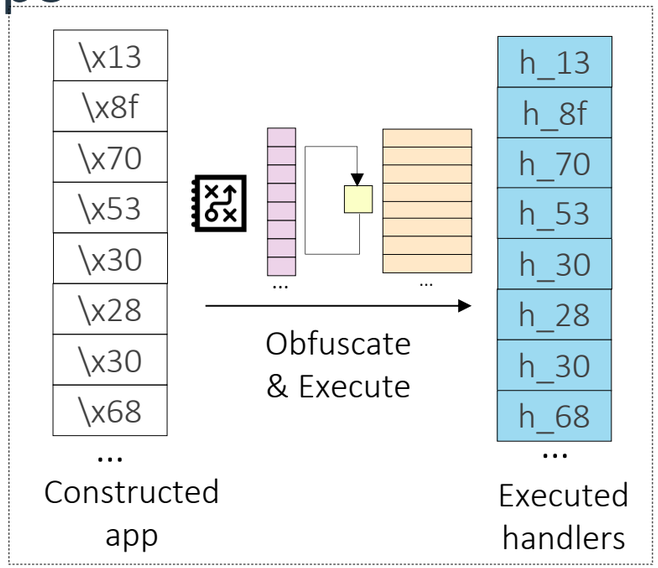
Learn the Mapping Relations

- Provided the above intuition, we can construct apps, obfuscate them and execute obfuscated programs, to learn the mapping relations between original Dalvik bytecode and executed handlers
- Then we can apply the learned rules to transform back executed handler information of other apps



Questions to Solve During the Learning Phase

1. Determine virtualized instructions for each function
2. Figure out relationship between virtualized instruction and handler address
3. Identify handlers by their content, so as to recognize each handler when executing in-the-wild apps



Learn mapping relations

Proposed Solution

1. Determine virtualized instructions for each function
- Hook and trace the register function

2. Figure out mapping relations between virtualized instruction and handler address
- Instrumentation to collect Dynamic Trace + analyze trace log to construct mapping

3. Identify handlers by their content, so as to recognize them in trace of other apps
- Generate Genetic Signature to identify each handler

1.Virtualization of Functions

```
.method protected onCreate(Landroid/os/Bundle;)V
    .registers 6

    .param p1, "savedInstanceState":Landroid/os/Bundle;

                            .line 20
    000014f8: 6f20 0100 5400          0000: invoke-super        {p0, p1}, Landroidx/activity/ComponentActivity;-
>onCreate(Landroid/os/Bundle;)V # method@0001
                            .line 21
    000014fe: 0740                    0003: move-object         v0, p0
    00001500: 1f00 0400               0004: check-cast          v0, Landroidx/activity/ComponentActivity; #
type@0004
    00001504: 6201 0c00               0006: sget-object         v1, Lcom/example/myapplication/
ComposableSingletons$MainActivityKt;->INSTANCE:Lcom/example/myapplication/ComposableSingletons$MainActivityKt; #
field@000c
    00001508: 6e10 2b00 0100          0008: invoke-virtual      {v1}, Lcom/example/myapplication/
ComposableSingletons$MainActivityKt;->getLambda-3$app_debug()Lkotlin/jvm/functions/Function2; # method@002b
    0000150e: 0c01                    000b: move-result-object  v1
```

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ComponentActivityKt.setContent$default(this, null, ComposableSingletons$MainActivityKt.INSTANCE.
m4078getLambda3$app_debug(), 1, null);
    }
```

Before obfuscation

```
public native void onCreate(Bundle bundle);
```

After obfuscation

```
__int64 __fastcall sub_C8C4(__int64 a1, __int64 a2)
{
  void *v3; // [xsp+0h] [xbp-70h] BYREF
  int v4; // [xsp+8h] [xbp-68h]
  __int128 *v5; // [xsp+10h] [xbp-60h]
  int *v6; // [xsp+18h] [xbp-58h]
  void *v7; // [xsp+20h] [xbp-50h]
  int v8; // [xsp+28h] [xbp-48h] BYREF
  __int16 v9; // [xsp+2Ch] [xbp-44h]
  __int128 v10[2]; // [xsp+30h] [xbp-40h] BYREF
  __int64 v11; // [xsp+50h] [xbp-20h]
  __int64 v12; // [xsp+58h] [xbp-18h]
  __int64 v13; // [xsp+68h] [xbp-8h]

  v13 = *(_QWORD *)(_ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2)) + 40);
  v4 = 61;
  v11 = 0LL;
  v12 = a2;
  v9 = 256;
  v3 = &unk_6CC8;
  memset(v10, 0, sizeof(v10));
  v8 = 0;
  v5 = v10;
  v6 = &v8;
  v7 = &unk_6D42;
  return vmInterpret(a1, &v3, &off_10720);
}
```

```
data:0000000000006CC8 E6 00 46 00 00 00 05 00 01 00+word_6CC8 DCW
data:0000000000006CC8 38 00 6F 00 6B 00 D9 01 2D 00+
data:0000000000006CC8 20 02 67 27 D3 30 1B 01 10 02+DCW 0x11E, 0,
data:0000000000006CC8 9F 10 1E 01 00 00 05 01 A1 02+DCW 0xE1, 1, 0
data:0000000000006CC8 01 01 06 00 E6 10 47 00 02 00+DCW 0x54D3, 0x
data:0000000000006D42 02                            unk_6D42 DCB
```

Virtualized instructions

1.Determine Virtualized Instructions for each Function

- Hook and trace the arguments passed to env->RegisterNatives(), to get signature of function and address of virtualized instructions

```
static int registerNativeMethods(JNIEnv* env)
{
    jclass clazz;
    clazz = env->FindClass("np/manager/Protect");
    if (clazz == NULL) {
        return JNI_FALSE;
    }
    static JNINativeMethod gMethods[] = {
    {"classesInit0", "(I)V", (void*)Jni_classesInit0},
    ...
    };
    if (env->RegisterNatives(clazz, gMethods, sizeof(gMethods) / sizeof(gMethods[0])) < 0) {
        return JNI_FALSE;
    }
    return JNI_TRUE;
}
```

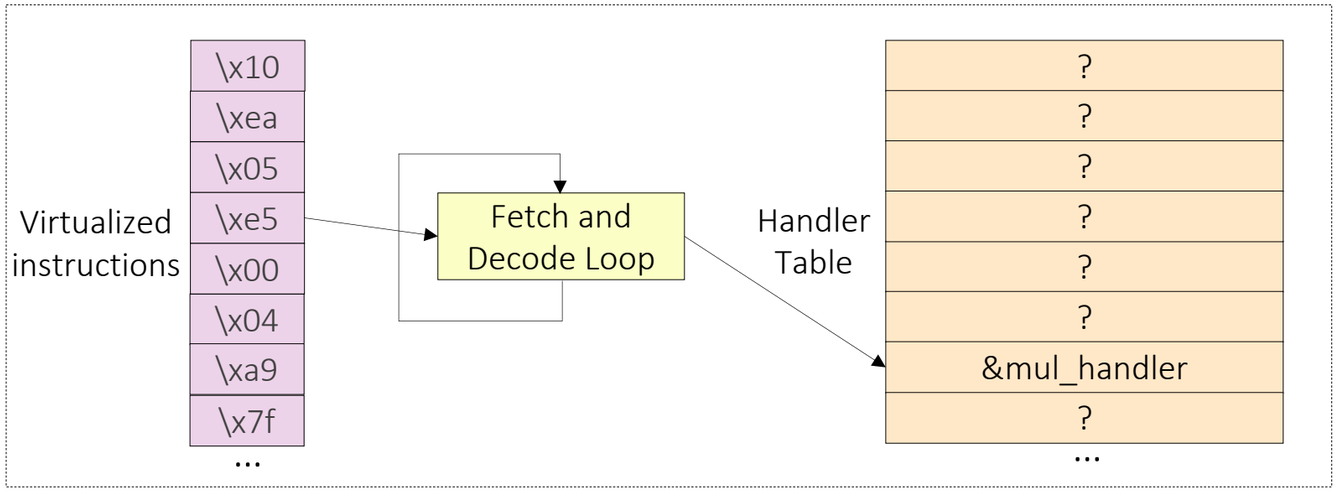
Hook

Name	Signature	Virtualized instructions
onCreate	(Landroid/os/Bundle;)V	\x12\x20\x00\x00T\x00D@\xd0\x00…
invoke	(Landroidx/compose/runtime/Composer;I)V	\x01\x00\x04\x00\xa7\x20\r\x00\x05\x00]\x00\x06…
…	…	…

Dumped Virtualized instructions

2. Mapping Relations between Virtualized Instruction and Handler Address

- There is need to observe execution flow [inside] function, so merely hook the entrance and exit of function is not enough
- Debugging is the first thing comes to mind, but it requires much tedious manual work, and coping with anti-debug mechanisms
- So we use Instrumentation + Trace way to construct the mapping



2. Tools and Frameworks for Instruction-level Instrumentation

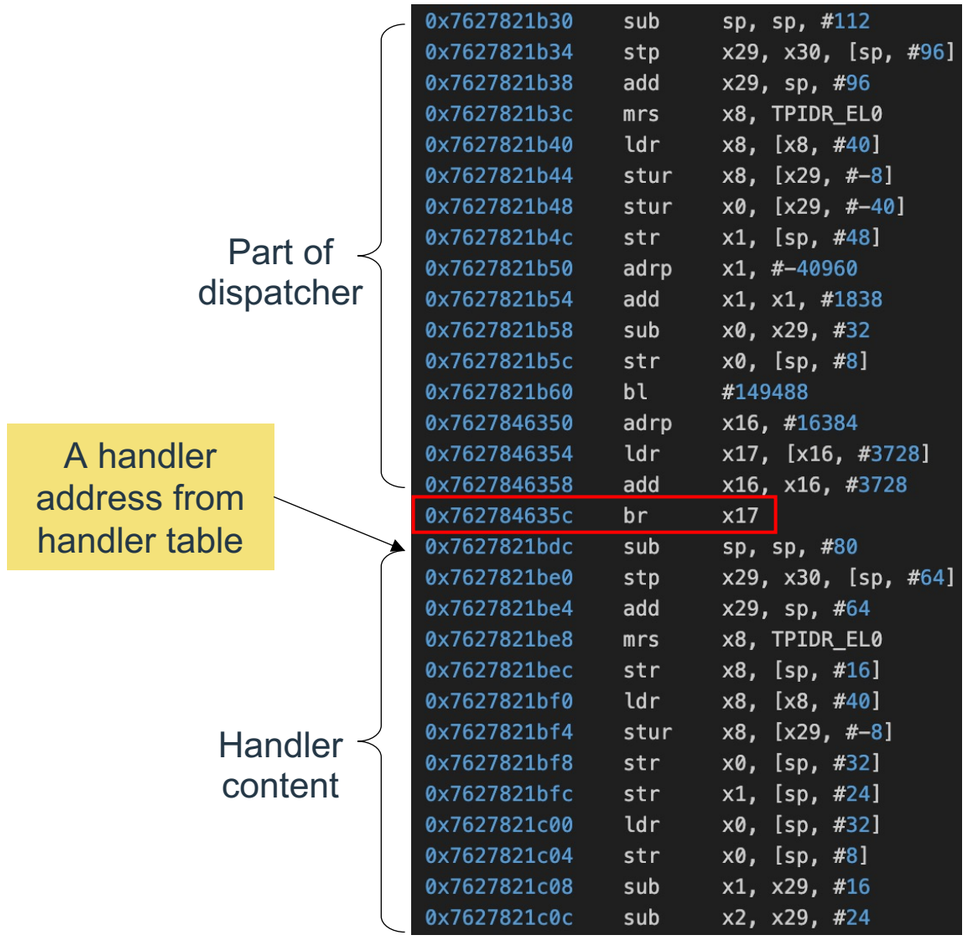
- DBI : Valgrind, DynamoRIO, QBDI, Intel Pin…
- Emulator: Unicorn, Unidbg…
- Frida-Stalker

Considerations
- Android support: exclude Intel Pin and Valgrind
- Environment supplement: necessary when using a simulator

QBDI or other instruction-level instrumentation frameworks can serve our purpose

2. Locating Dispatcher and Handler Table

- Dispatcher firstly loads an address(of handler) to a register, and then uses “br” to jump to that address
- The address points to code segment, but itself is stored in data segment, or more precisely, the handler table
- The target of the jump is a handler, and after that the content, or instruction sequence of that handler is executed



2. Mapping Virtualized Instruction and Handler Address

- Each run of the above function builds a mapping between a virtualized instruction and a handler

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
00000000  94 00 10 00 00 00 78 00 92 00 05 00 bb 00 12 00  ......x.........
00000010  ba 00 9f 00 13 00 92 00 10 00 bb 00 12 00 94 10  ................
00000020  2a 00 00 00 be 00 9a 01 07 00 94 20 11 00 01 00  *.......... ....
00000030  be 00 18 00 13 00 71 10 0e 00 00 00 be 00 98 00  ......q.........
00000040  2d 00 32 10 2b 00 00 00 78 00 ba 00              -.2.+...x...
```

Dumped Virtualized instructions

```
.data.rel.ro:000000000003C018 30 76 01 00 00 00 00 00       off_3C018 DCQ sub_17630
.data.rel.ro:000000000003C020 1C D0 01 00 00 00 00 00       DCQ loc_1D01C
.data.rel.ro:000000000003C028 C0 96 01 00 00 00 00 00       DCQ loc_196C0
.data.rel.ro:000000000003C030 60 C7 01 00 00 00 00 00       DCQ loc_1C760
.data.rel.ro:000000000003C038 AC 86 01 00 00 00 00 00       DCQ loc_186AC
.data.rel.ro:000000000003C040 64 B0 01 00 00 00 00 00       DCQ loc_1B064
.data.rel.ro:000000000003C048 68 76 01 00 00 00 00 00       DCQ sub_17668
.data.rel.ro:000000000003C050 80 7E 01 00 00 00 00 00       DCQ loc_17E80
.data.rel.ro:000000000003C058 34 84 01 00 00 00 00 00       DCQ loc_18434
.data.rel.ro:000000000003C060 D8 AC 01 00 00 00 00 00       DCQ loc_1ACD8
```

handler table

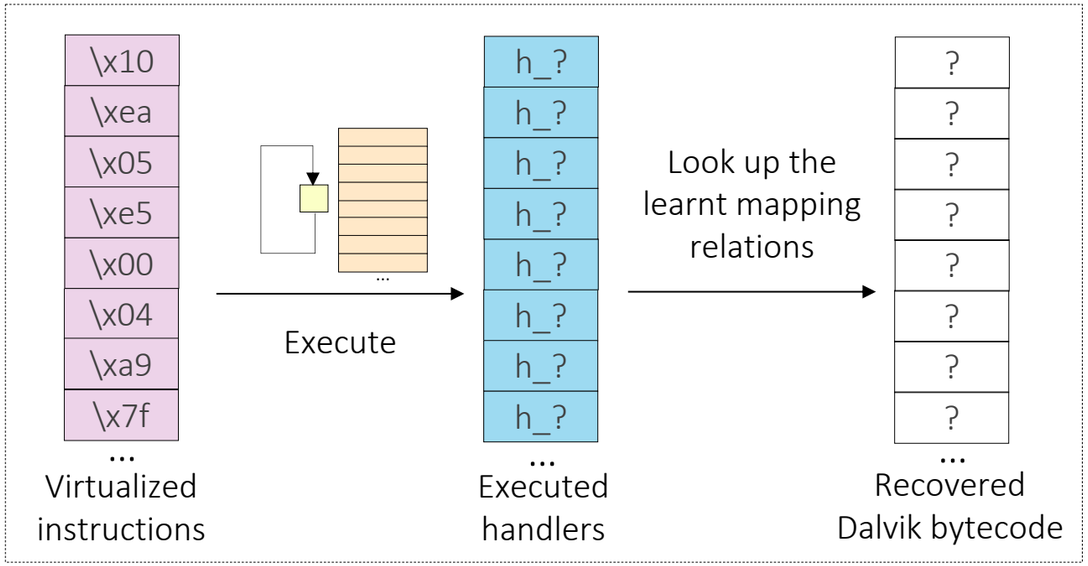
3. Need to Identify Each Handler

- When unpacking in-the-wild apps that original dex file is unavailable, the order of handlers is random, which means handler address only cannot identify a handler
- As a result, during the learning phase, we need to create ”identity” with handler’s content for each handler.

```
0x7627821b54    add     x1, x1, #1838
0x7627821b58    sub     x0, x29, #32
0x7627821b5c    str     x0, [sp, #8]
0x7627821b60    bl      #149488
0x7627846350    adrp    x16, #16384
0x7627846354    ldr     x17, [x16, #3728]
0x7627846358    add     x16, x16, #3728
0x762784635c    br      x17
0x7627821bdc    sub     sp, sp, #80
0x7627821be0    stp     x29, x30, [sp, #64]
0x7627821be4    add     x29, sp, #64
0x7627821be8    mrs     x8, TPIDR_EL0
0x7627821bec    str     x8, [sp, #16]
0x7627821bf0    ldr     x8, [x8, #40]
0x7627821bf4    stur    x8, [x29, #-8]
0x7627821bf8    str     x0, [sp, #32]
0x7627821bfc    str     x1, [sp, #24]
0x7627821c00    ldr     x0, [sp, #32]
0x7627821c04    str     x0, [sp, #8]
0x7627821c08    sub     x1, x29, #16
0x7627821c0c    sub     x2, x29, #24
```

Handler content

How do we know this is h_50?

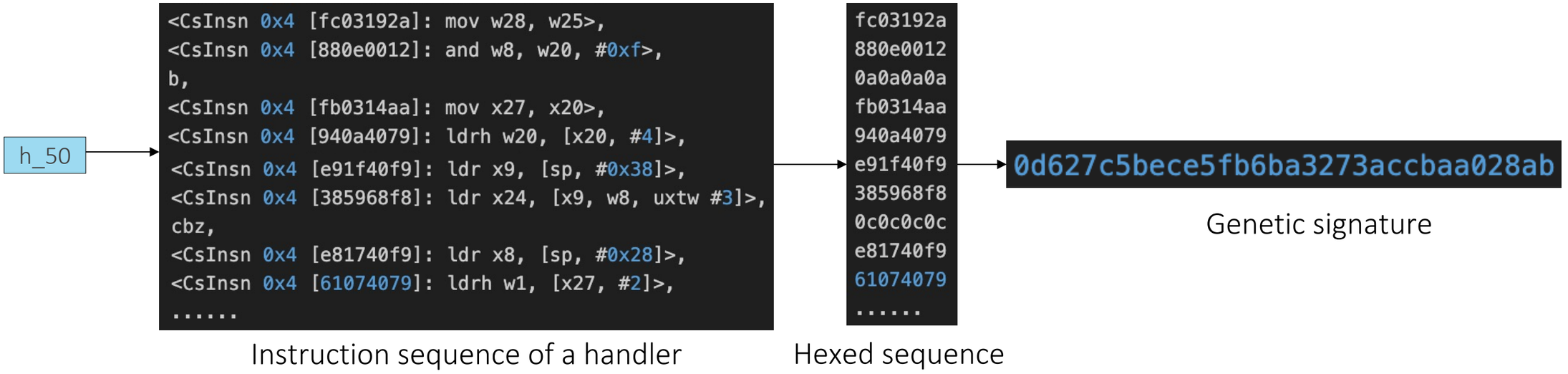


Apply learnt mapping relation to unpack apps

3. Genetic Signature of Handler

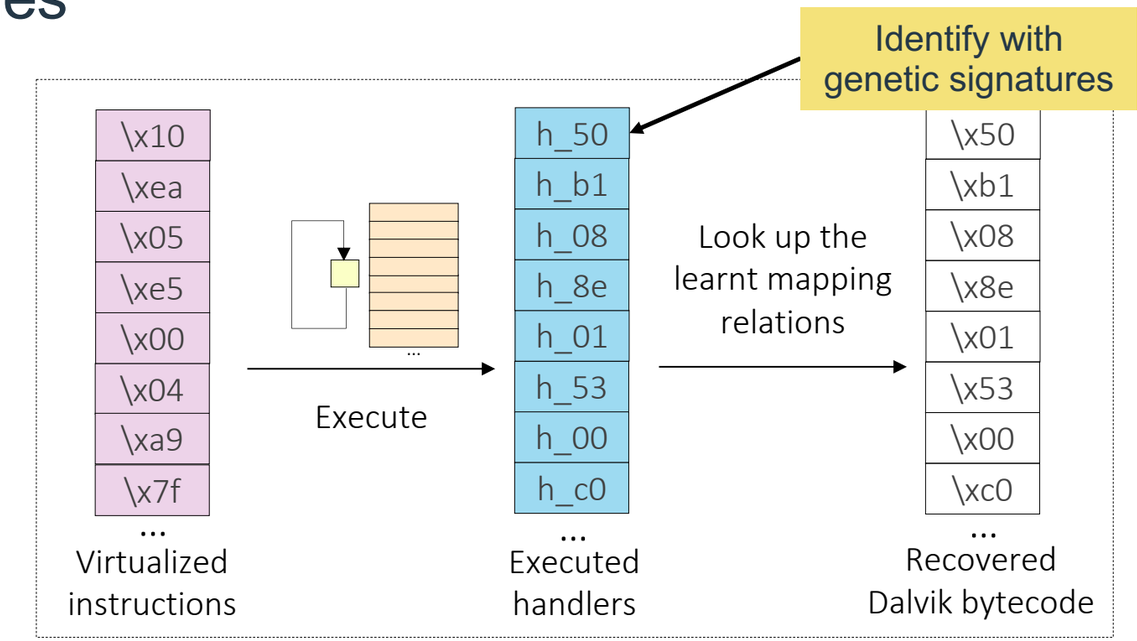
We generally use “hash(hex(instruction sequence))” as genetic signature of a handler, but this signature needs to undergo the following processing steps:

- Truncate the sequence to only include the part before the 'br' instruction.
- Replace instructions that have different corresponding machine codes across different programs with a fixed sequence. This kind of instructions include jump instructions(e.g. b, bl, cbz) and PC-register-relative instructions(e.g. adrp), etc.



Put it Together: Reversing In-the-wild Obfuscated App

- Execute in-the-wild apps to get genetic signatures corresponding to executed virtualized instructions, and then just apply learnt mapping relations to recover them into Dalvik bytecodes



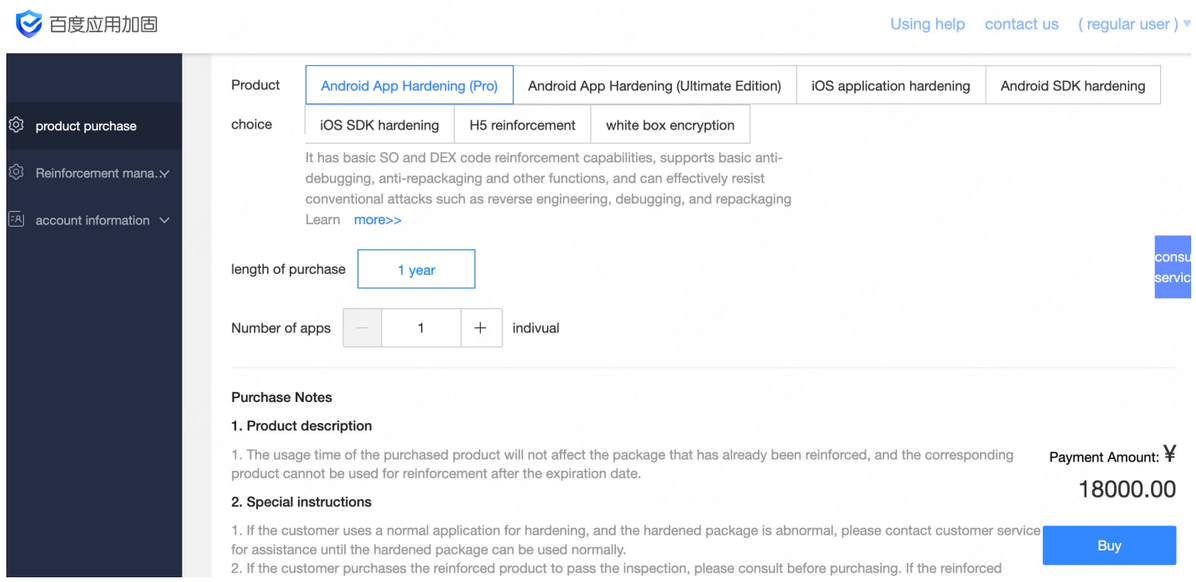
Apply learnt mapping relation to unpack apps

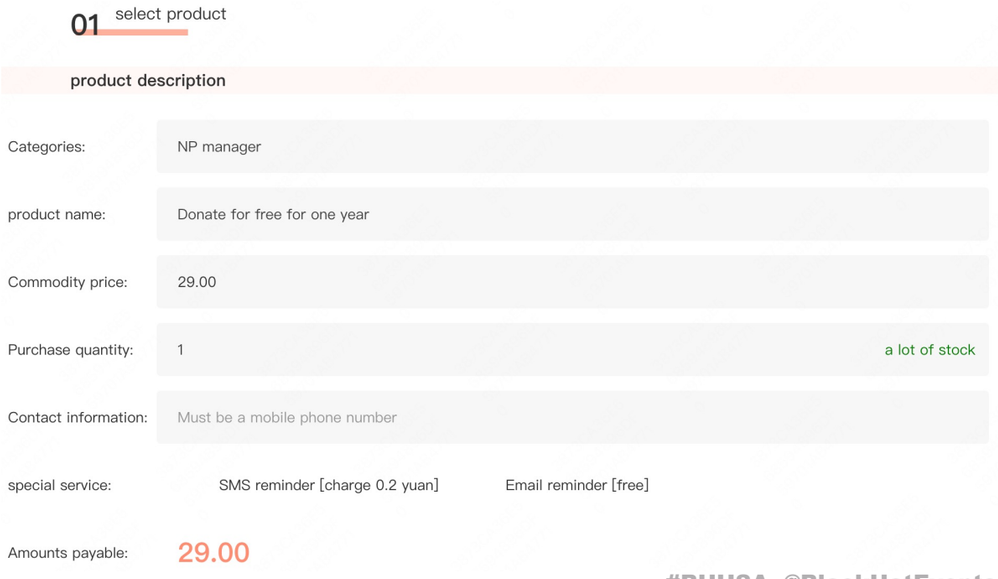
Agenda

- Intro & Background Story
- The Rhino Bytecode Case and its Reversing
- More General Scenario of VM-Packed Programs
- Unpacking under General Scenario
- Insights & Conclusion

Usage of Inexpensive Packer in Obfuscation of Android Malware

- We have noticed a tendency of malwares using inexpensive VM-based packers
- One of the most famous commercial packers charge ¥18000 (about $2516) per year, while some inexpensive packers only charge About $4 per year.
- This seems to attract price-sensitive malware authors. A significant portion of recently seen VM-protected malware samples are obfuscated with inexpensive packers





Another Common Obfuscation Technique on Android

- Hiding the Dex data and dynamically releasing it into memory during execution is another widely-used obfuscation technique.
- For this kind of obfuscation, existing unpacking tools search for and extract the Dex data from the memory.
- However, these tools are unable to unpack apps that are protected by VM obfuscation, as the original Dalvik bytecode is never placed into the memory.

Conclusion and Takeaways

In conclusion, we propose a two-fold methodology for Unpacking Android apps with VM-based obfuscation:

- For specific type of obfuscation with Rhino bytecode, since it is open-source, we analyze the VM and reconstruct AST to recover source from bytecode
- For more general scenario, we introduce a method through gathering execution trace and using genetic signatures to learn mapping relationship between original bytecode and handler. The learnt relations are then applied to recover semantics of in-the-wild VM-obfuscated apps.

Reference

- https://web.archive.org/web/20230531155247/https://blog.autojs.org/2022/08/24/encryption/
- https://swarm.ptsecurity.com/how-we-bypassed-bytenode-and-decompiled-node-js-bytecode-in-ghidra/
- https://github.com/QBDI/QBDI
- https://www4.comp.polyu.edu.hk/~csxluo/Parema.pdf

Thanks!