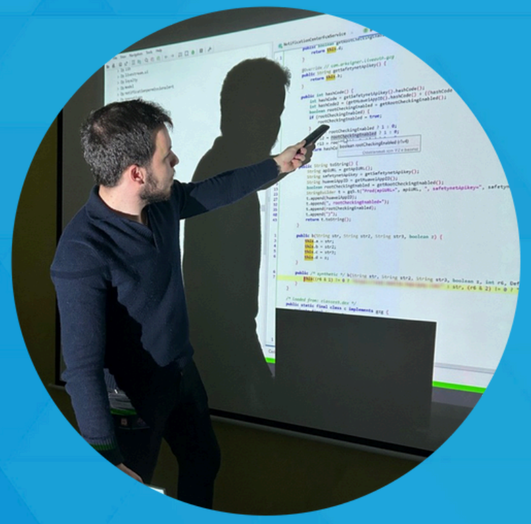
Devirtualizing VM-Based Obfuscation in Android

#r2con2025

Ahmethan Gultekin

/in/ahmethangultekin

BYTERIA



Ahmethan **Gultekin**

- founder of **Byteria**
- mobile researching - reverse engineering
- public talks / private trainings

BYTERIA

Agenda:

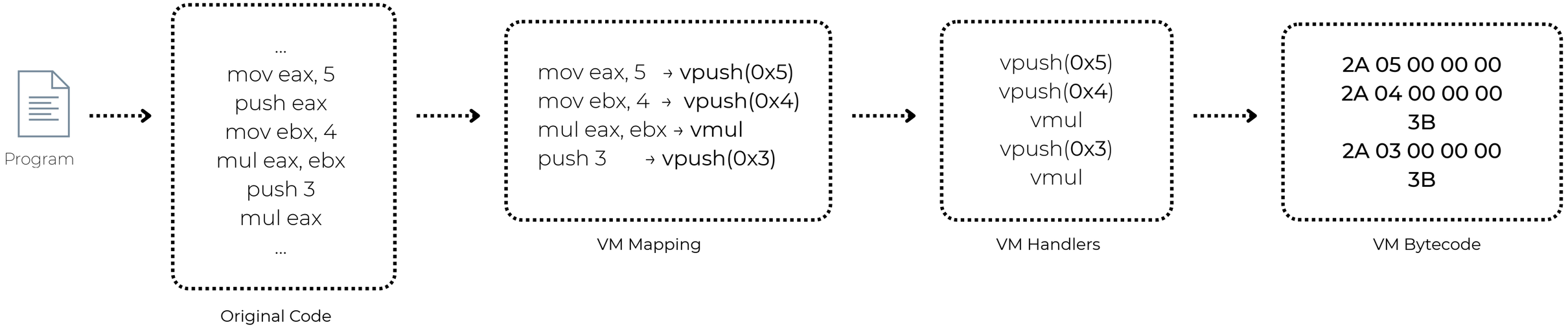
- Introduction to VM fundamentals
- VM Hardening Techniques
- Devirtualizing VM-based Protection (PairIP Protection)

Understanding Virtual Machine Fundamentals

Virtual Machine Basics:

- A virtual machine takes normal instructions and converts them into custom **opcodes**.
- Each opcode is processed by a small function called a handler, while a dispatcher runs the loop that fetches, decodes, and executes them.
- This extra layer changes how the code looks and makes reverse engineering much more difficult.

From Instructions to VM Bytecode



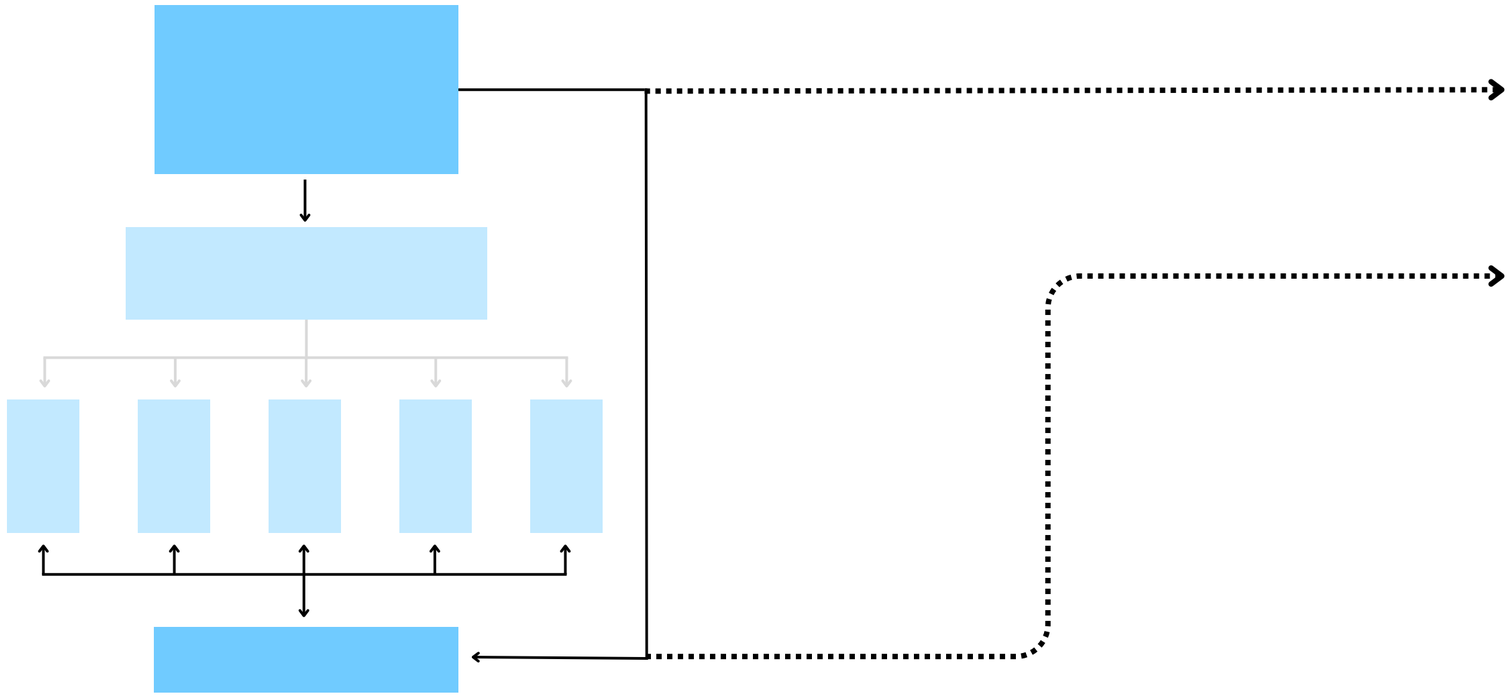
Bytecode Analysis

```
2A 05 00 00 00   2A 04 00 00 00   3B   2A 03 00 00 00   3B
```

- 1 byte opcode + optional operand

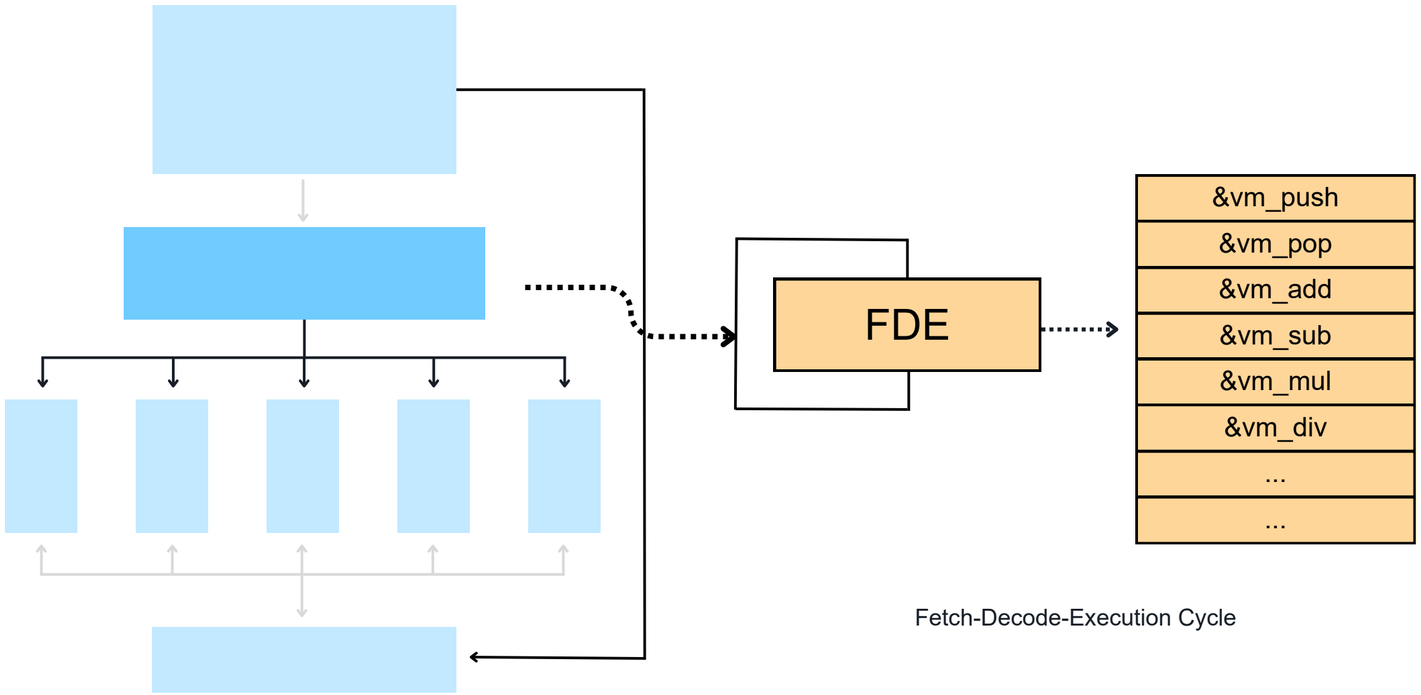
Bytecode	Decoded VM Assembly	High-Level Code
2A 05	vpush(5)	int eax = 5
2A 04	vpush(4)	int ebx = 4
3B	vmul(5x4)	int res = eax * ebx
2A 03	vpush(3)	res = res * 3
3B	vmul(20*3)	// final result is 60

Virtual Machine Basics:



- **VM Entry**: entry point into the virtual machine, initializes context/state, jumps into the interpreter loop
- **VM Exit**: termination of the virtual machine, loop ends, context cleaned, returns control to native code
- **VM Dispatcher**: decode step, reads current opcode, looks up the handler in the table, jumps via indirect **call**/**blr**
- **VM Handler**: code block that implements the semantics of a single opcode (e.g., push, add, jmp)

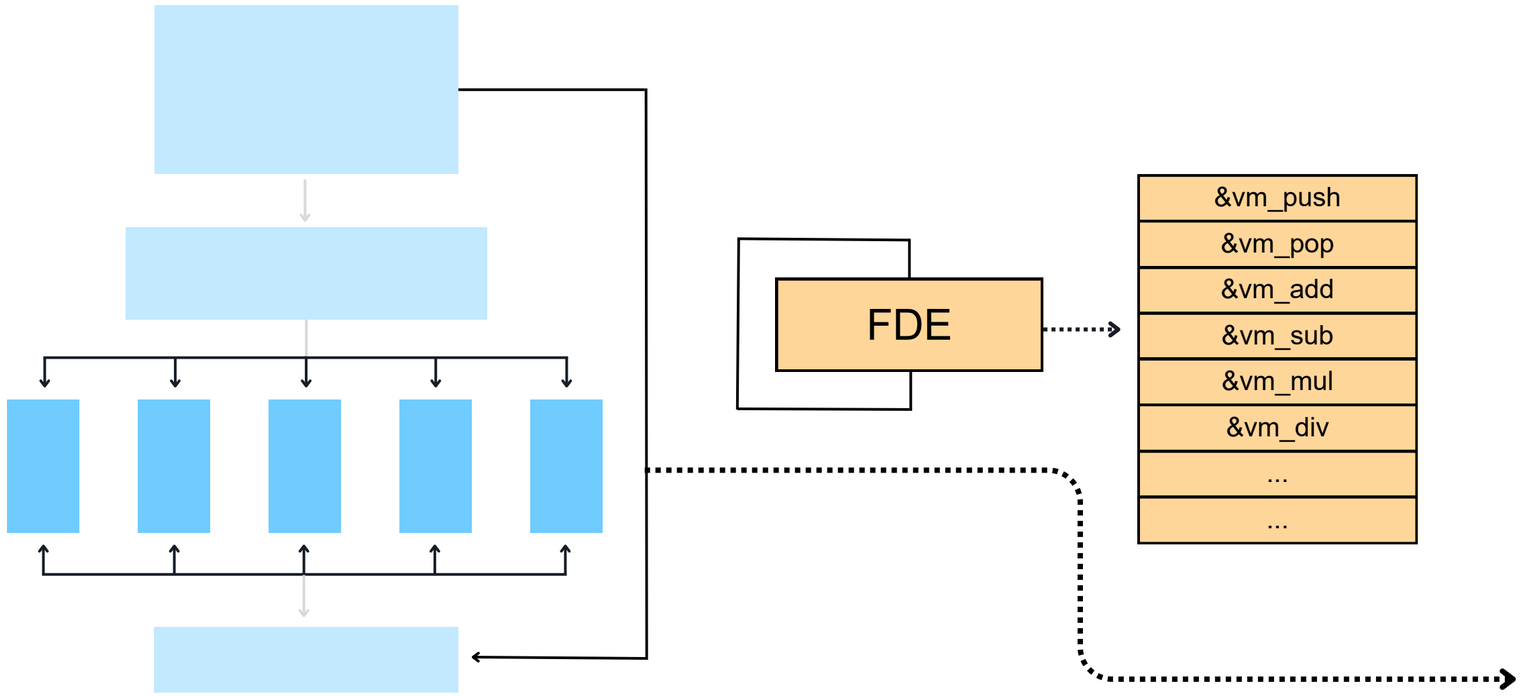
Virtual Machine Basics:



Fetch-Decode-Execution Cycle

- **VM Entry**: entry point into the virtual machine, initializes context/state, jumps into the interpreter loop
- **VM Exit**: termination of the virtual machine, loop ends, context cleaned, returns control to native code
- **VM Dispatcher**: decode step, reads current opcode, looks up the handler in the table, jumps via indirect **call**/**blr**
- **VM Handler**: code block that implements the semantics of a single opcode (e.g., push, add, jmp)

Virtual Machine Basics:



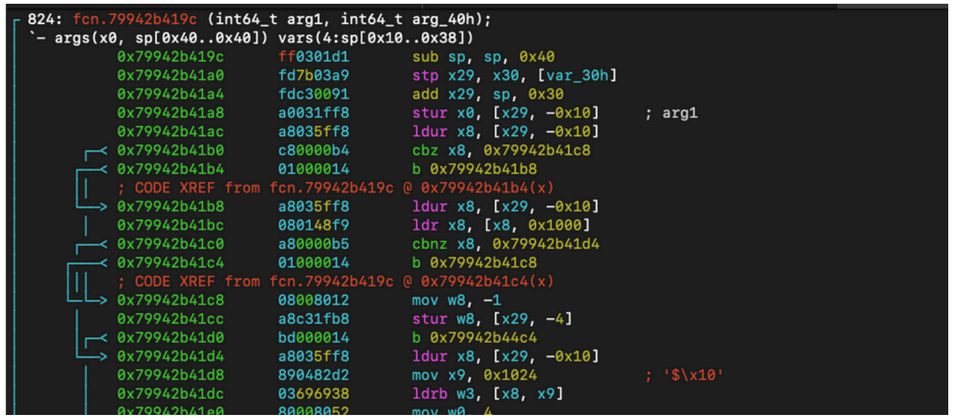
- each handler represent the 1 opcode. (per opcode)

- **VM Entry**: entry point into the virtual machine, initializes context/state, jumps into the interpreter loop
- **VM Exit**: termination of the virtual machine, loop ends, context cleaned, returns control to native code
- **VM Dispatcher**: decode step, reads current opcode, looks up the handler in the table, jumps via indirect **call/blr**
- **VM Handler**: code block that implements the semantics of a single opcode (e.g., push, add, jmp)

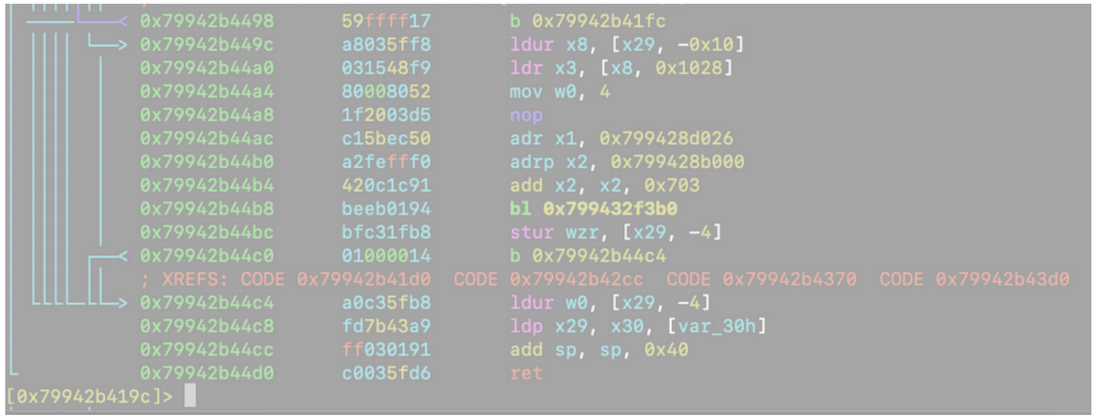
Virtual Machine Basics:

- **Function Prologue**: reserve space on stack, save frame pointer & return address ,setup new stack frame

- Function Epilogue: prepare return value, restore saved registers, clean stack & ret back



prologue concept
(0x79942b419c~0x79942b41a8)



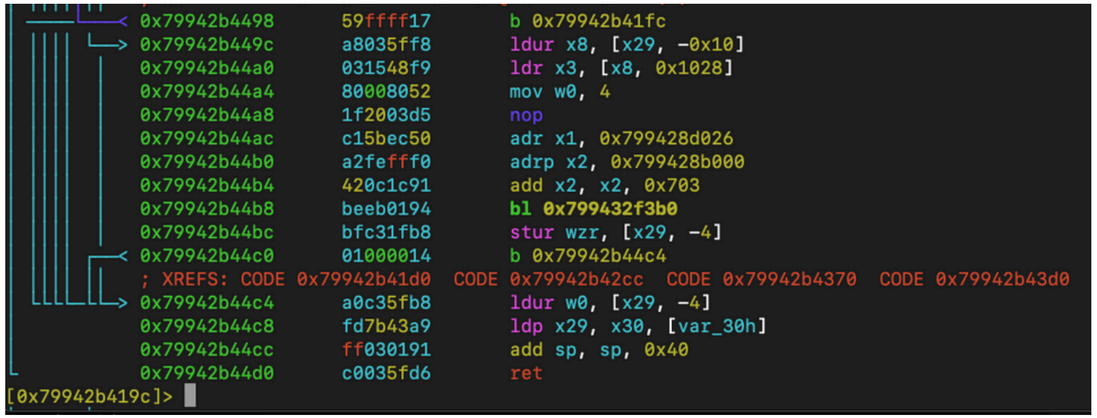
epilogue concept
(0x79942b44c4~0x79942b44d0)

Virtual Machine Basics:

- Function Prologue: reserve space on stack, save frame pointer & return address ,setup new stack frame

- **Function Epilogue**: prepare return value, restore saved registers, clean stack & ret back

prologue concept
(0x79942b419c~0x79942b41a8)

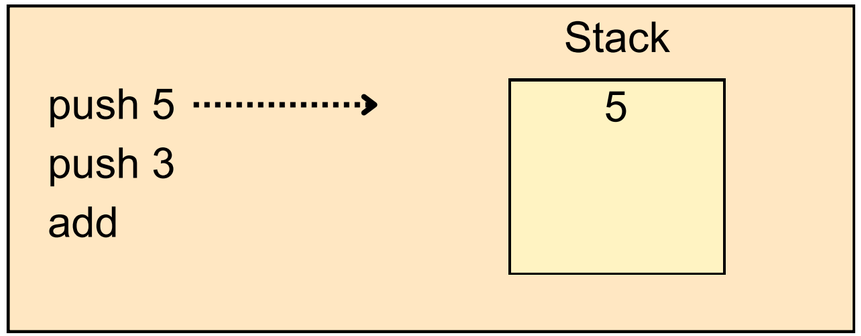


epilogue concept
(0x79942b44c4~0x79942b44d0)

Virtual Machine Types:

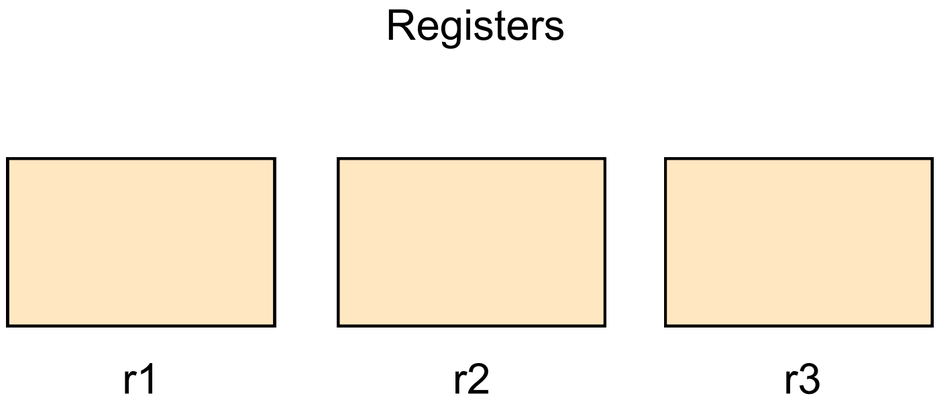
Stack-Based VM:

- Operands are pushed and popped from a stack.
- Instructions are simple and compact then Register-Based VM



Register-Based VM:

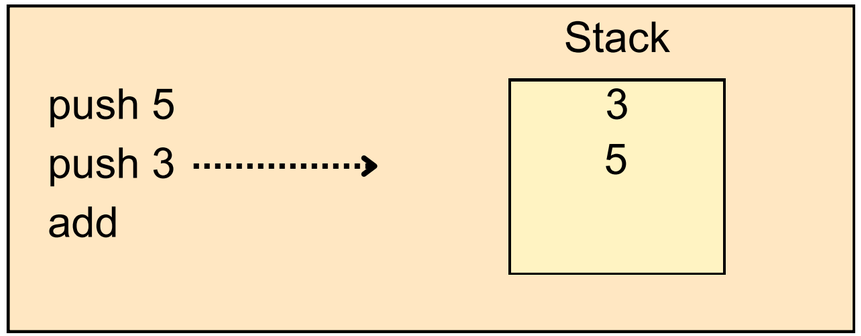
- Operands are pushed and popped from a stack.
- Instructions are simple and compact than Register-Based VM



Virtual Machine Types:

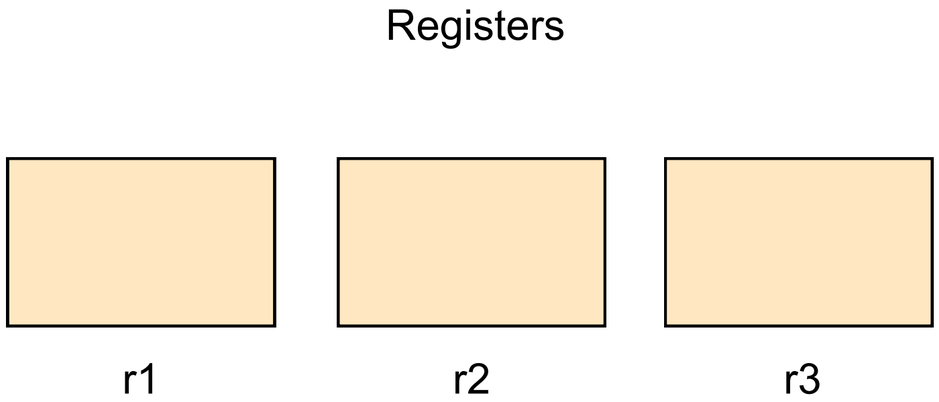
Stack-Based VM:

- Operands are pushed and popped from a stack.
- Instructions are simple and compact then Register-Based VM



Register-Based VM:

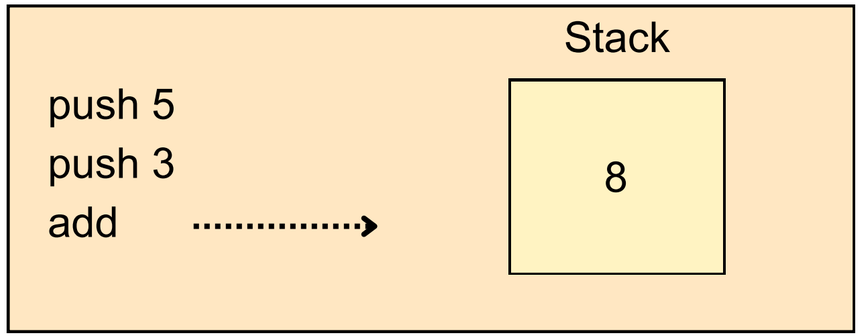
- Operands are pushed and popped from a stack.
- Instructions are simple and compact than Register-Based VM



Virtual Machine Types:

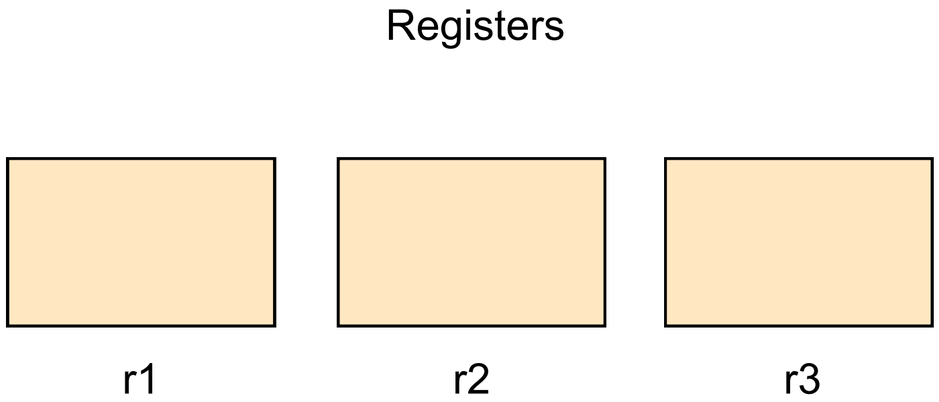
Stack-Based VM:

- Operands are pushed and popped from a stack.
- Instructions are simple and compact then Register-Based VM



Register-Based VM:

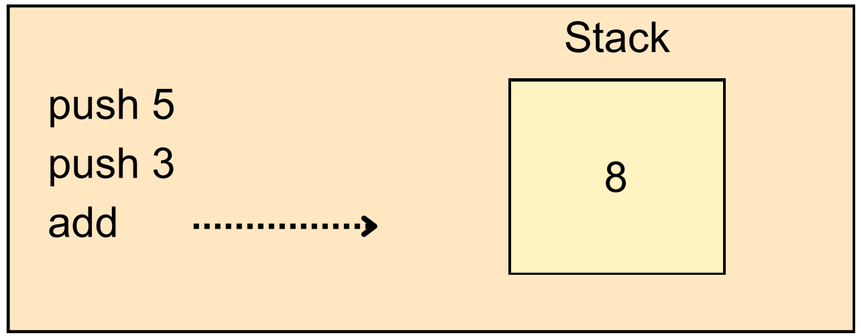
- Operands are pushed and popped from a stack.
- Instructions are simple and compact than Register-Based VM



Virtual Machine Types:

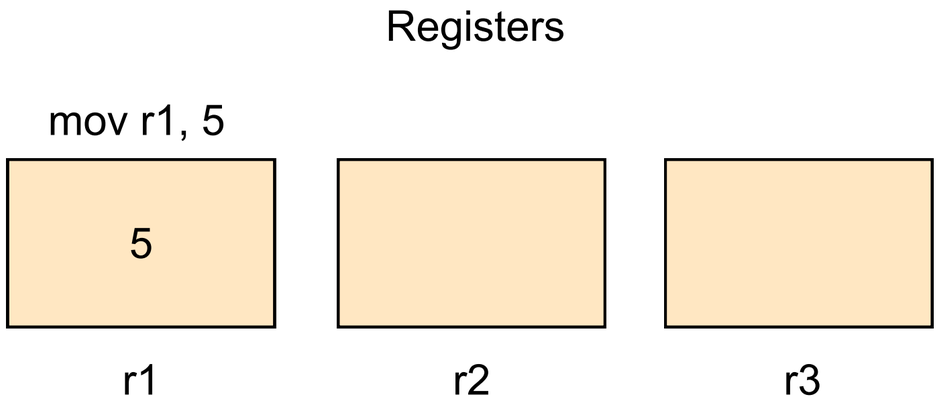
Stack-Based VM:

- Operands are pushed and popped from a stack.
- Instructions are simple and compact then Register-Based VM



Register-Based VM:

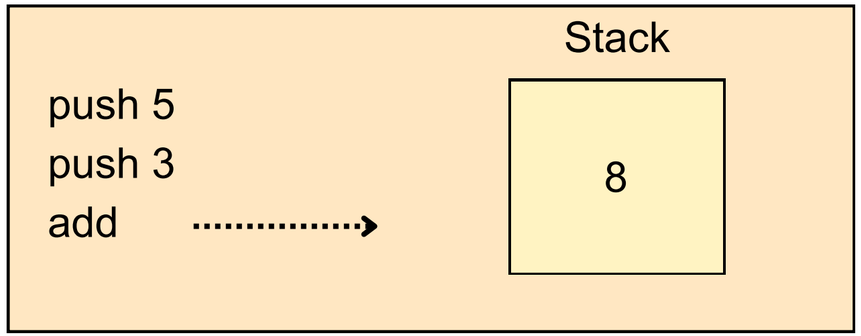
- Operands are pushed and popped from a stack.
- Instructions are simple and compact than Register-Based VM



Virtual Machine Types:

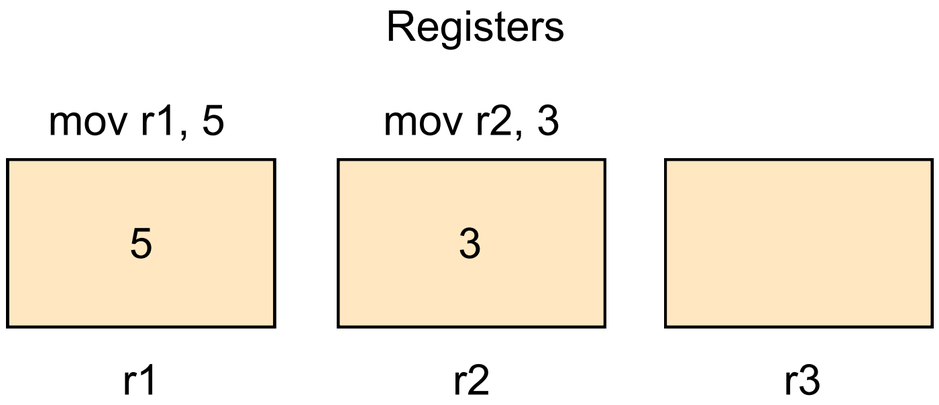
Stack-Based VM:

- Operands are pushed and popped from a stack.
- Instructions are simple and compact then Register-Based VM



Register-Based VM:

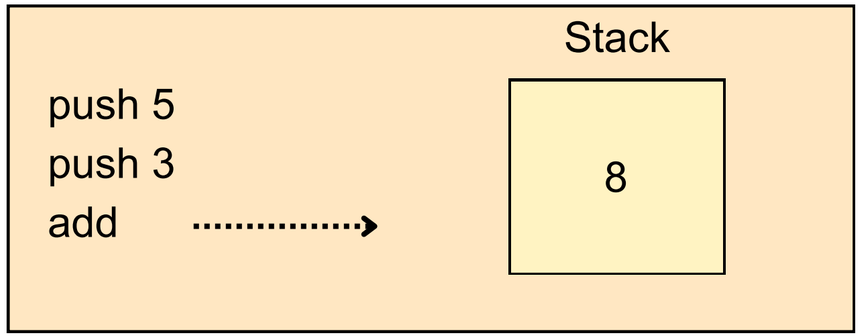
- Operands are pushed and popped from a stack.
- Instructions are simple and compact than Register-Based VM



Virtual Machine Types:

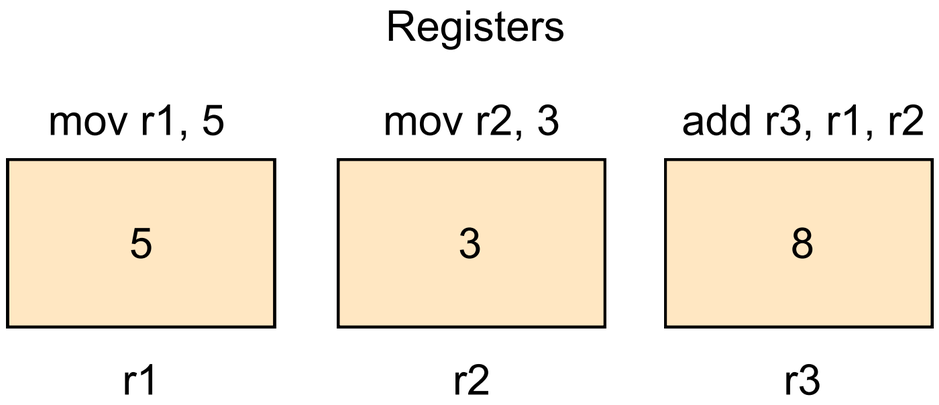
Stack-Based VM:

- Operands are pushed and popped from a stack.
- Instructions are simple and compact then Register-Based VM



Register-Based VM:

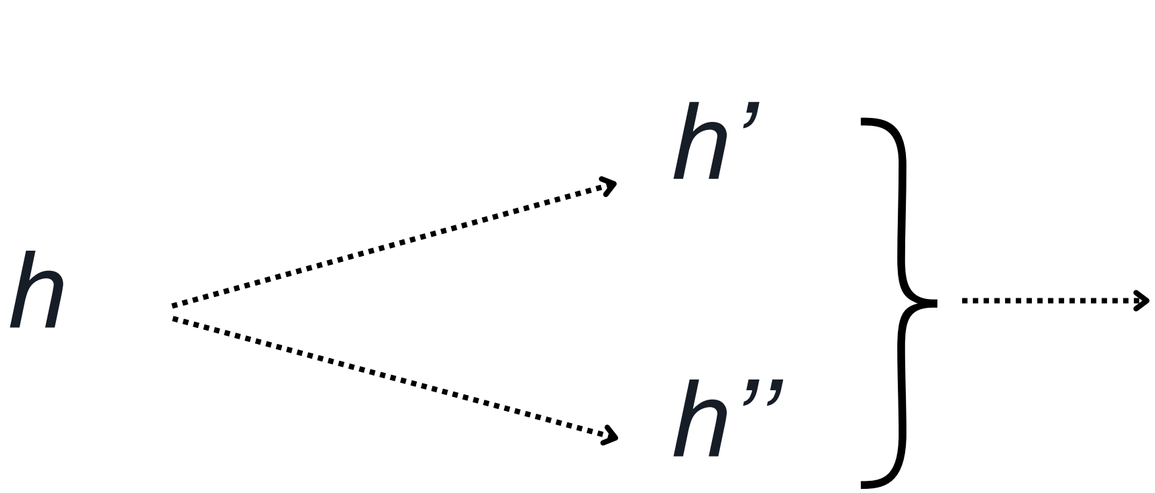
- Operands are pushed and popped from a stack.
- Instructions are simple and compact than Register-Based VM



Virtual Machine Hardening

Duplicating VM Handlers:

- Duplicate the same handler to shuffle the VM table



Handler Table

vm_handler_0
h’
vm_handler_1
h’’
vm_handler_2
h
...
...

Control Flow Flattening

- Hides the real execution order by breaking normal program structure into a dispatcher-based flow
- It introduces artificial branches and loops, making reverse engineering significantly more complex

Normal Program CFG

Flattened Program CFG

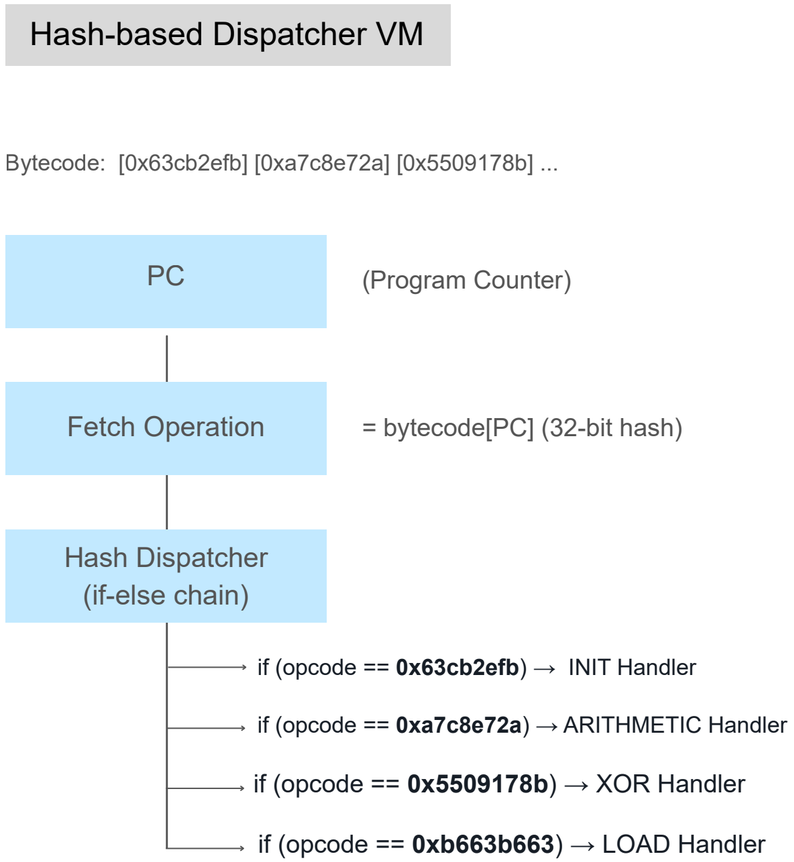
Hash-based Dispatching

- Uses FNV-1 32-bit hashes as opcodes instead of sequential values
- Makes static analysis harder (cannot guess next opcode)
- Prevents trivial patching (must recompute hash)

Example:

Normal VM: 0x01, 0x02, 0x03 (predictable)
Hash-based Dispatching VM: 0x63cb2efb, 0x781245d6, 0xb663b663 (hash values)

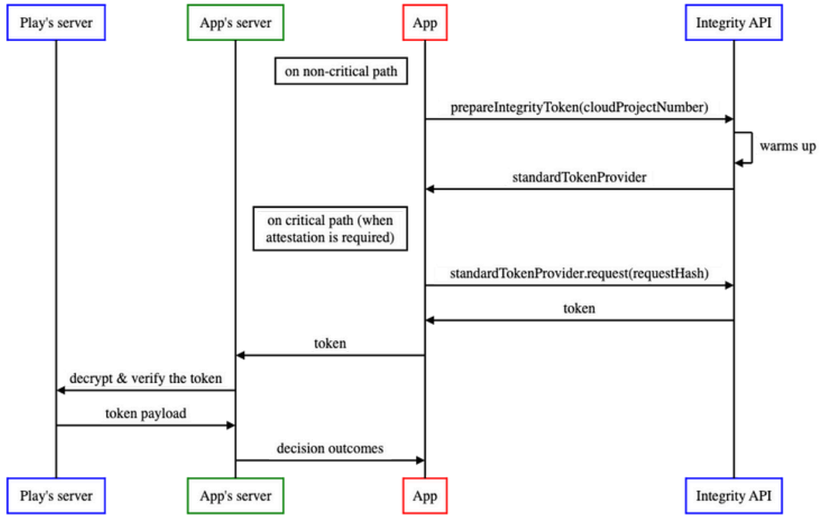
we will see its use in real life on the next pages :)

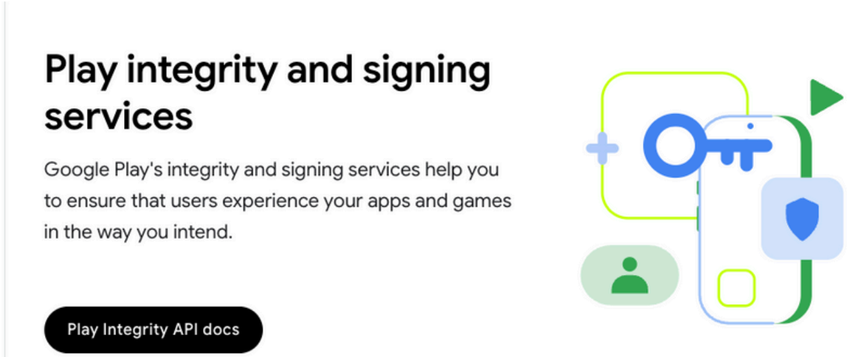


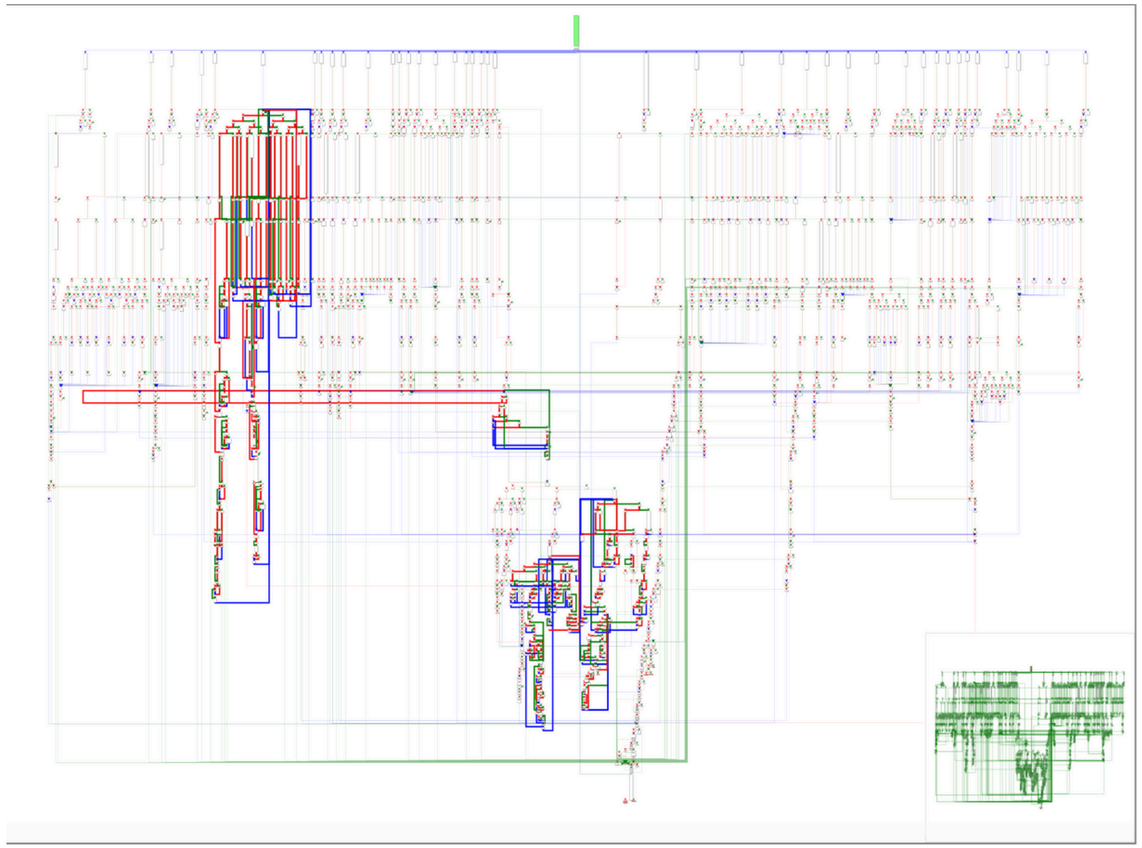
Real-Life Examples
(PairIP Protection)

What is the PairIP?

- PairIP has similar mechanisms to Safetynet as basic checks. It subjects your device to a security check by performing an integrity check on the device
- VM-based infrastructure
- Every handler processes a VM instruction in a fixed pipeline:
- *parse operands → execute logic → hash/verify (e.g. FNV-1)*
- heavy obfuscated VM methods, encrypted bytecodes, hash-based dispatcher VM



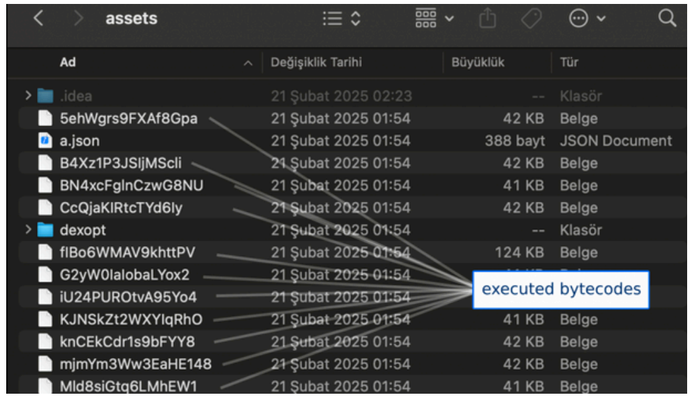




executeVM method CFG

Technical Analysis

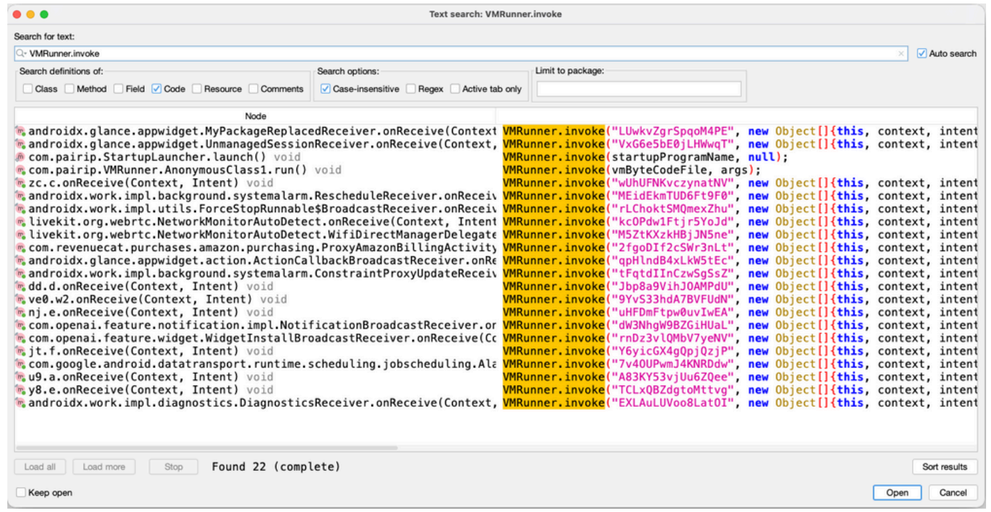
- bytecode files storing in ***assets/*** folder



- bytecodes are executed by **invoke()** method on Java side. basic call-tree like this;

```
invoke() → getVmByteCode() → executeVM() → return executeVM;
```

- the encrypted byte codes are first decrypted in the *executeVM* method. *executeVM* method is registered using the **RegisterNatives** JNI method.

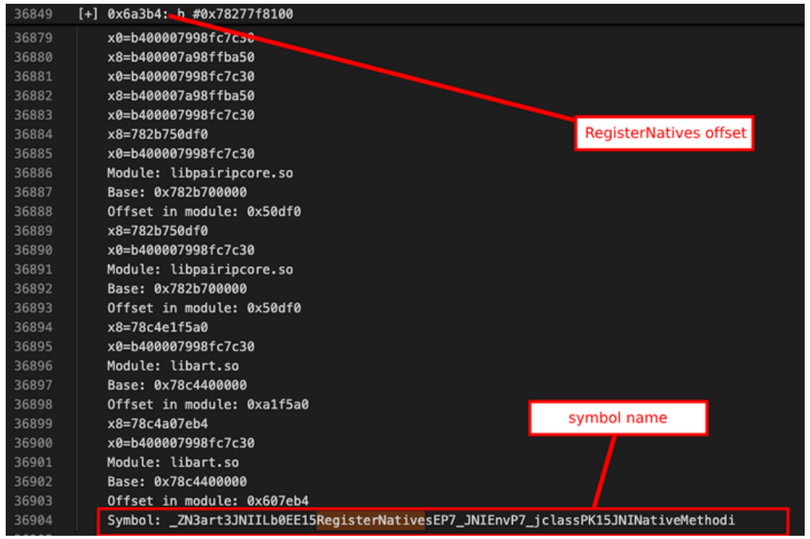


- executeVM method is not exist in so library (*because implementing with RegisterNatives call*). if we want to look closer, we'll need to pull it from runtime memory.

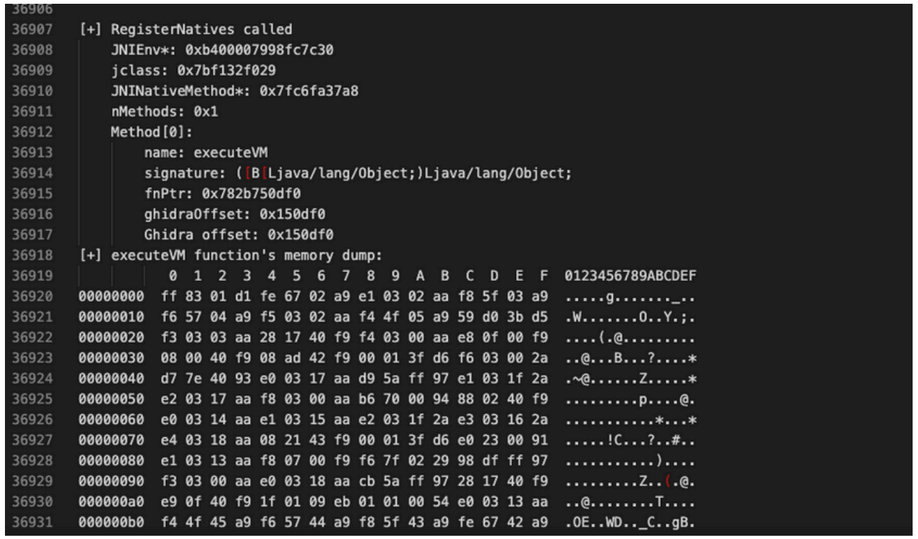
- all bytecode files have .IAP magic. but we don't know what it means.

Technical Analysis - Frida Step

enumerateSymbols → get **RegisterNatives** offset → Interceptor → hook *do_dlopen* →
if *libpairipcore.so* loaded → hook *RegisterNatives* → get **executeVM** offset



RegisterNatives offset



```
[+] RegisterNatives called
    JNIEnv*: 0xb400007998fc7c30
    jclass: 0x7bf132f029
    JNINativeMethod*: 0x7fc6fa37a8
    nMethods: 0x1
    Method[0]:
        name: executeVM
        signature: ([B)Ljava/lang/Object;)Ljava/lang/Object;
        fnPtr: 0x782b750df0
        ghidraOffset: 0x150df0
        Ghidra offset: 0x150df0
[+] executeVM function's memory dump:
             0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
00000000  ff 83 01 d1 fe 67 02 a9 e1 03 02 aa f8 5f 03 a9  .....g......._..
00000010  f6 57 04 a9 f5 03 02 aa f4 4f 05 a9 59 d0 3b d5  .W.......O..Y.;.
00000020  f3 03 03 aa 28 17 40 f9 f4 03 00 aa e8 0f 00 f9  ....(.@.........
00000030  08 00 40 f9 08 ad 42 f9 00 01 3f d6 f6 03 00 2a  ..@...B...?....*
00000040  d7 7e 40 93 e0 03 17 aa d9 5a ff 97 e1 03 1f 2a  .~@......Z.....*
00000050  e2 03 17 aa f8 03 00 aa b6 70 00 94 88 02 40 f9  .........p....@.
00000060  e0 03 14 aa e1 03 15 aa e2 03 1f 2a e3 03 16 2a  ...........*...*
00000070  e4 03 18 aa 08 21 43 f9 00 01 3f d6 e0 23 00 91  .....!C...?..#..
00000080  e1 03 13 aa f8 07 00 f9 f6 7f 02 29 98 df ff 97  ...........)....
00000090  f3 03 00 aa e0 03 18 aa cb 5a ff 97 28 17 40 f9  .........Z..(.@.
000000a0  e9 0f 40 f9 1f 01 09 eb 01 01 00 54 e0 03 13 aa  ..@........T....
000000b0  f4 4f 45 a9 f6 57 44 a9 f8 5f 43 a9 fe 67 42 a9  .OE..WD.._C..gB.
```

executeVM() offset found with RegisterNatives

Technical Analysis - r2 Step

PairIP has heavy VM-obfuscated structure.

- dynamic opcode table (changes at runtime)
- bytecode verification with FNV-1 hash
- executing encrypted bytecodes (in assets/ folder)
- address obfuscation (virtual memory)
- string obfuscation

and a lot of VM hardening techniques..

let’s take a closer look..

Technical Analysis - Bytecode Integrity Check

for the bytecode integrity checks, PairIP uses **FNV-1** hash.

hashes each instruction during execution, compares with expected opcode → if mismatch, VM crashes

prevents bytecode modification attacks

formula: (pdg output)

```
uVar36 = uVar36 * 0x100000001b3 ^ param_3;
```

disadvantages for RE:

- Hard to identify opcodes
- Random-looking hashes
- Obfuscated flow

```
[0x00052ddc]> pd 10
           0x00052ddc      a16484d2       mov x1, 0x2325              ; '%#'
           0x00052de0      4184b0f2       movk x1, 0x8422, lsl 16
           0x00052de4      290140f9       ldr x9, [x9]
           0x00052de8      819cd3f2       movk x1, 0x9ce4, lsl 32
           0x00052dec      417ef9f2       movk x1, 0xcbf2, lsl 48
       ┌─< 0x00052df0      d0010034       cbz w16, 0x52e28
       │   0x00052df4      d101114a       eor w17, w14, w17
       │   0x00052df8      ef031faa       mov x15, xzr
       │   0x00052dfc      200ac81a       udiv w0, w17, w8
       │   0x00052e00      11c4081b       msub w17, w0, w8, w17
[0x00052ddc]>
```

FNV-1 constant initialization

```
[0x0004d7d4]> s 0x52cac
[0x00052cac]> pd 3
           0x00052cac      663680d2       mov x6, 0x1b3
           0x00052cb0      e1031f2a       mov w1, wzr
           0x00052cb4      0620c0f2       movk x6, 0x100, lsl 32
[0x00052cac]>
```

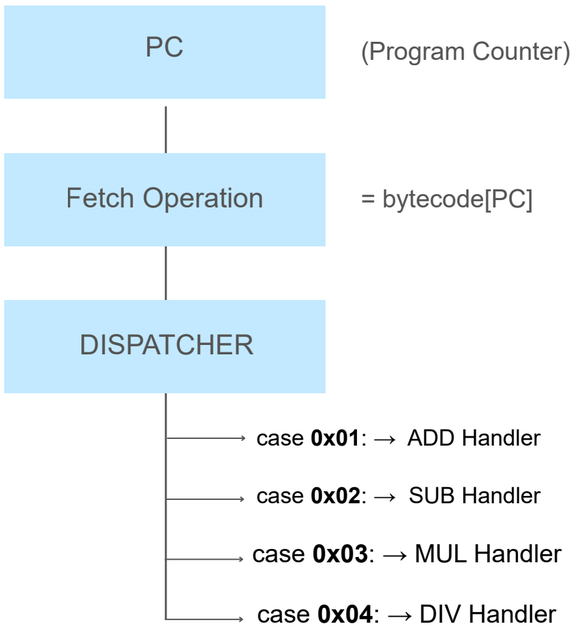
FNV-1 prime initialization

```
[0x00052e0c]> pd 10
           ; CODE XREF from fcn.00051534 @ +0x18f0(x)
       ┌─> 0x00052e0c      217c039b       mul x1, x1, x3
       ¦   0x00052e10      226aaf38       ldrsb x2, [x17, x15]
       ¦   0x00052e14      0f7c4093       sxtw x15, w0
       ¦   0x00052e18      10060071       subs w16, w16, 1
       ¦   0x00052e1c      00040011       add w0, w0, 1
       ¦   0x00052e20      210002ca       eor x1, x1, x2
       └─< 0x00052e24      41ffff54       b.ne 0x52e0c
           ; CODE XREF from fcn.00051534 @ +0x18bc(x)
           0x00052e28      cc010c4a       eor w12, w14, w12
           0x00052e2c      8f09c81a       udiv w15, w12, w8
           0x00052e30      ecb1081b       msub w12, w15, w8, w12
```

FNV-1 hash loop

Technical Analysis - Bytecode Integrity Check

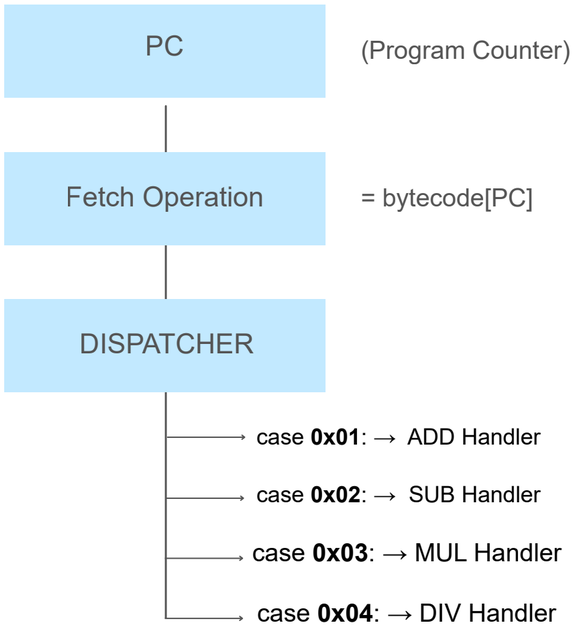
Traditional VM:



Technical Analysis - Bytecode Integrity Check

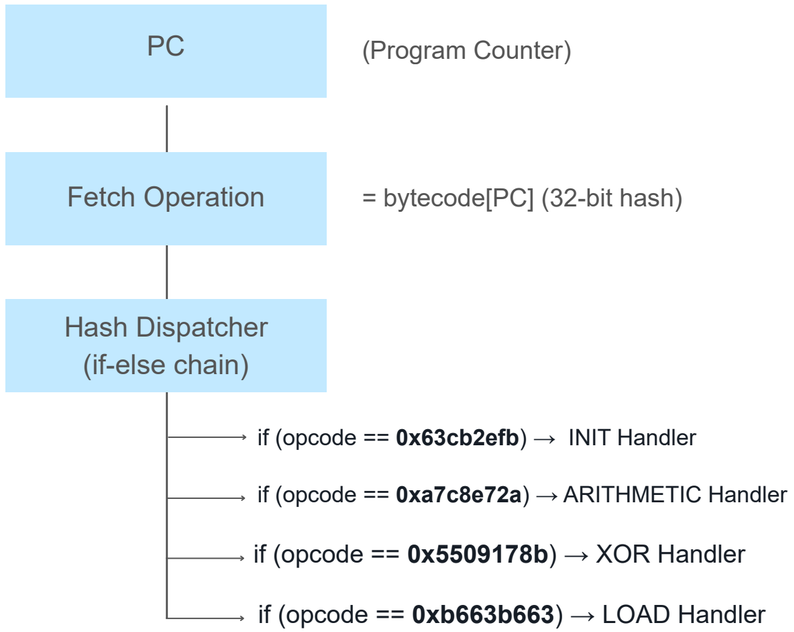
Traditional VM:

Bytecode: [01] [02] [03] [04] [05] ...



Hash-based Dispatcher VM

Bytecode: [0x63cb2efb] [0xa7c8e72a] [0x5509178b] ...



Technical Analysis - Bytecode Integrity Check

for traditional VM

```
switch(opcode) {
    case 0x01: do_add(); break;
    case 0x02: do_sub(); break;
    case 0x03: do_mul(); break;
}
```

for Hash-based dispatcher VM

```
opcode = read_from_bytecode();  // like 0x758fee41 hash value
if (opcode == 0x758fee41) {
    do_call();
}
else if (opcode == 0x63cb2efb) {
    do_init();
}
else if (opcode == 0x9ca4d828) {
    do_goto();
}

//...
```

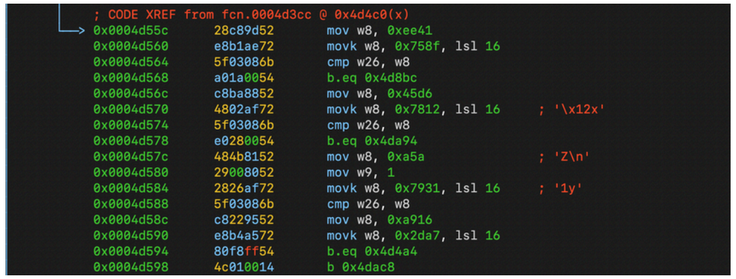
Technical Analysis - Opcode Analysis

- 32-bit immediates cannot fit in a single instruction

solution: split into two 16-bit parts using mov + movk pattern

mov wX, *LOW_16* → Load lower 16 bits (clears register)
movk wX, *HIGH_16* → Keep and merge upper 16 bits (lsl 16)

- opcodes are hashes. used directly for comparison
  example: 0x758fee41 = FNV1("CALL instruction bytes")

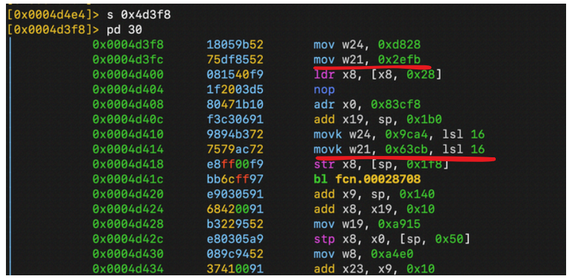


mov w8, 0xee41 => 0x0000_ee41
movk w8, 0x758f, lsl 16 => 0x758f_0000

w8: ***0x758fee41***

mov w8, 0x45d6 => 0x0000_45d6
movk w8, 0x7812, lsl 16 => 0x7812_0000

w8: ***0x781245d6***



*mov w21, 0x2fb => **0x0000_2efb***
*movk w21, 0x63cb, lsl 16 => **0x63cb_0000***

w21: ***0x63cb2efb***

*mov w24, 0xd828 => **0x0000_d828***
*movk w24, 0x9ca4, lsl 16 => **0x9ca4_0000***

w24: ***0x9ca4d828***

⚠ these opcodes are not static values, changes at runtime.

Technical Analysis - Opcode Analysis

- let's see how to work Google's opcodes

```
[[0x0004da94]> s 0x4d56c
[[0x0004d56c]> pd 10
            0x0004d56c      c8ba8852        mov w8, 0x45d6
            0x0004d570      4802af72        movk w8, 0x7812, lsl 16
            0x0004d574      5f03086b        cmp w26, w8
        ┌<  0x0004d578      e0280054        b.eq 0x4da94
        │   0x0004d57c      484b8152        mov w8, 0xa5a
        │   0x0004d580      29008052        mov w9, 1
```

mov w8, 0x45d6 => 0x0000_45d6
movk w8, 0x7812, lsl 16 => 0x7812_0000

---

w8: ***0x781245d6***

after that, branch to 0x4da94 handler.

0x781245d6 opcode's handler. (0x4da94)

```
[[0x0004d56c]> s 0x4da94
[[0x0004da94]> pd 10
        ¦   0x0004da94      e14f40b9        ldr w1, [sp, 0x4c]
        ¦   0x0004da98      e0830291        add x0, sp, 0xa0
        ¦   0x0004da9c      473f0094        bl 0x5d7b8
        ¦   0x0004daa0      484b8152        mov w8, 0xa5a               ; 'Z\n'
        ¦   0x0004daa4      1f000072        tst w0, 1
        ¦   0x0004daa8      e903162a        mov w9, w22
        ¦   0x0004daac      2826af72        movk w8, 0x7931, lsl 16     ; '1y'
        ¦   0x0004dab0      a812881a        csel w8, w21, w8, ne
        ¦┌< 0x0004dab4      7cfeff17        b 0x4d4a4
        ¦   0x0004dab8      887a8a52        mov w8, 0x53d4
```

- *Load arguments from stack (arg1, arg2)*
- *Call native function at 0x5d7b8 using bl instruction*
- *Check return value with tst*
- *Set next opcode (0x7931a5a or w21 based on condition)*
- *Return to dispatcher to continue VM execution*

⚠ these opcodes are not static values, changes at runtime.

Technical Analysis - Opcode Analysis

for now, we know PairIP VM uses **mov/movk** pattern for opcode construction

maybe we can automate this using the r2pipe API?

Parse disassembly JSON, find patterns, construct opcodes

the pattern is:

*mov w8, **0x2efb** ← LOW 16 bits*
*movk w8, **0x63cb**, lsl 16 ← HIGH 16 bits*

---

*Result: **0x63cb2efb** ← 32-bit opcode*

⚠ these opcodes are not static values, changes at runtime.

```python
def extract_opcodes(binary_path, vm_offset=0x4d3cc):

    r2 = r2pipe.open(binary_path, flags=['-2'])
    r2.cmd('aaa')
    r2.cmd(f's 0x{vm_offset:x}')
    r2.cmd('af')

    disasm = r2.cmdj('pdfj')

    opcodes = {}

    if disasm and 'ops' in disasm:
        ops = disasm['ops']
        i = 0
        while i < len(ops):
            op = ops[i]

            # Look for: mov wX, #imm
            if op.get('type') == 'mov' and 'disasm' in op:
                disasm_str = op['disasm']
                mov_match = re.search(r'mov\s+w(\d+),\s+(?:#)?0x([0-9a-f]+)', disasm_str)

                if mov_match and i + 1 < len(ops):
                    reg = mov_match.group(1)
                    low = int(mov_match.group(2), 16)

                    next_op = ops[i + 1]
                    if next_op.get('type') == 'mov' and 'disasm' in next_op:
                        next_disasm = next_op['disasm']
                        movk_match = re.search(
                            rf'movk\s+w{reg},\s+(?:#)?0x([0-9a-f]+),\s+lsl\s+(?:#)?16',
                            next_disasm
                        )

                        if movk_match:
                            high = int(movk_match.group(1), 16)
                            opcode = (high << 16) | low
                            addr = op['offset']

                            opcodes[opcode] = addr
                            i += 2
                            continue

            i += 1

    dispatcher_reg = None
    if disasm and 'ops' in disasm:
```

Technical Analysis - Opcode Analysis

Result:

```
(venv) ➜  r2con2025 python3 r2_opcode_extractor_r2pipe.py libpairipcore.so.dump

Extracted Opcodes:
[765962518, 1426659211, 1972366912, 1972366913, 2014463446, 2628048936, 2814961450, 2921873583, 3059988067, 3079271189, 3428144084]

Total: 11 opcodes
Dispatcher register: w8
```

⚠ these opcodes are not static values, changes at runtime.

Technical Analysis - Opcode Analysis

Opcode	Name	Description
0x1deaa4e0	INITIAL_STATE	Initial dispatcher value at VM entry
0x2da7a916	SETUP	VM initialization and environment setup
0x5509178b	XOR_OP	XOR operation with constant 0x7f55
0x63cb2efb	INIT	Entry point opcode (first executed)
0x758fee41	CLEANUP	Clean up resources and free memory
0x781245d6	CALL_2	Function call with 2 arguments
0x9ca4d828	GOTO	Unconditional jump to bytecode address
0xa7c8e72a	ARITHMETIC	Complex arithmetic (add/sub/mul/div)
0xb663b662	BOUNDARY_CHK	Loop boundary check
0xb663b663	LOAD_BYTECODE	Load bytecode from PC and increment
0xb789f315	LOOP_CHECK	Loop counter check (continue/break)
0xbcef0d0f	STACK_PUSH	Push value onto VM stack
0xcc5553d4	EXIT	Exit VM and return to JNI
0xfb7da2e6	STACK_POP	Pop value from VM stack

⚠ these opcodes are not static values, changes at runtime.

Technical Analysis - String Encryption

Google, using same magic key for all bytecode files. (not .IAP)
this key length is 5803 and it using for string encryption process.

it can be found **heuristic scan** technique.

- starts with: ab 16 2b c0 2f 44 37 b6...
- same across ALL PairIP-protected apps
- **XOR**-based: byte[i] ⊕ key[i % 256]

Decrypt:
string[i] ⊕ key[i mod 256] → plaintext
len = (data_len ⊕ key_len) + 2

this key is constantly updated. so if you try to decrypt new PairIP version binary strings with the oldest keys, you will most likely fail.

```
→  r2con2025 xxd -s 0x1752f -l 256 data.vmcode | head -16
0001752f: ab16 2bc0 2f44 37b6 7f51 318f 3940 1311  ..+./D7..Q1.9@..
0001753f: 2aec 38dd 37da 4f22 5f54 9700 1d3b a654  *.8.7.O"_T...;.T
0001754f: 60ad bf50 8c86 fe67 c9c2 afaf 5fba 71c1  `..P...g...._.q.
0001755f: 9a36 1b71 2eb6 4333 0aa5 e5ea f920 59f1  .6.q..C3..... Y.
0001756f: 7411 1325 f287 d8b6 8ecd a823 b36f d84e  t..%.......#.o.N
0001757f: 52e8 bed9 c1a0 6ac2 2856 77d9 43fc 926b  R.....j.(Vw.C..k
0001758f: 0c23 f8a9 68b8 f7d4 24ac ad2d 8857 928a  .#..h...$..-.W..
0001759f: b379 cd59 65d9 aba8 d193 8791 6ff5 63eb  .y.Ye.......o.c.
000175af: a005 c7d4 c63f 80b9 f5a3 d07c 7f4f 0a27  .....?.....|.O.'
000175bf: e1f5 be98 b5d1 d950 5f6c 4918 78a2 16f8  .......P_lI.x...
000175cf: f7ab 03f0 af5f e8f8 f6ce cc0d d2b3 b466  ....._.........f
000175df: 4ff2 a93c d70e 0405 a485 b526 cfbc 16d3  O..<.......&....
000175ef: fe0b b8fa b1fb 34f8 1676 9296 e3ee 97f1  ......4..v......
000175ff: c1ad 15e3 0f18 3a5e 2ffe 141a dd1b e971  ......:^/......q
0001760f: 11c8 c3aa f2a0 ca22 7d91 cdc8 015f 3848  ......."}...._8H
0001761f: e96a bef4 1a42 cc6d a229 5c1d 66b6 381c  .j...B.m.)\.f.8.
```

```
→  r2con2025 dd if=data.vmcode bs=1 skip=$((0x1752f)) count=256 2>/dev/null | xxd -p | tr -d '\n'

ab162bc02f4437b67f51318f394013112aec38dd37da4f225f5497001d3ba65460adbf508c86fe67c9c2afaf5fba71c19a361b712eb643
52e8bed9c1a06ac2285677d943fc926b0c23f8a968b8f7d424acad2d8857928ab379cd5965d9aba8d19387916ff563eba005c7d4c63f80
f7ab03f0af5fe8f8f6cecc0dd2b3b4664ff2a93cd70e0405a485b526cfbc16d3fe0bb8fab1fb34f816769296e3ee97f1c1ad15e30f183a
e96abef41a42cc6da2295c1d66b6381c
```

```
→  r2con2025 python3 << 'EOF'
data = open('data.vmcode', 'rb').read()
key = data[0x1752f:0x1752f+256]
print("KEY = b'\\", end='')
for i, byte in enumerate(key):
    print(f'\\x{byte:02x}', end='')
    if (i+1) % 16 == 0 and i < 255:
        print("\\\n", end='')
print("'")
EOF
KEY = b'\\xab\x16\x2b\xc0\x2f\x44\x37\xb6\x7f\x51\x31\x8f\x39\x40\x13\x11\
\x2a\xec\x38\xdd\x37\xda\x4f\x22\x5f\x54\x97\x00\x1d\x3b\xa6\x54\
\x60\xad\xbf\x50\x8c\x86\xfe\x67\xc9\xc2\xaf\xaf\x5f\xba\x71\xc1\
\x9a\x36\x1b\x71\x2e\xb6\x43\x33\x0a\xa5\xe5\xea\xf9\x20\x59\xf1\
\x74\x11\x13\x25\xf2\x87\xd8\xb6\x8e\xcd\xa8\x23\xb3\x6f\xd8\x4e\
\x52\xe8\xbe\xd9\xc1\xa0\x6a\xc2\x28\x56\x77\xd9\x43\xfc\x92\x6b\
\x0c\x23\xf8\xa9\x68\xb8\xf7\xd4\x24\xac\xad\x2d\x88\x57\x92\x8a\
\xb3\x79\xcd\x59\x65\xd9\xab\xa8\xd1\x93\x87\x91\x6f\xf5\x63\xeb\
\xa0\x05\xc7\xd4\xc6\x3f\x80\xb9\xf5\xa3\xd0\x7c\x7f\x4f\x0a\x27\
\xe1\xf5\xbe\x98\xb5\xd1\xd9\x50\x5f\x6c\x49\x18\x78\xa2\x16\xf8\
\xf7\xab\x03\xf0\xaf\x5f\xe8\xf8\xf6\xce\xcc\x0d\xd2\xb3\xb4\x66\
\x4f\xf2\xa9\x3c\xd7\x0e\x04\x05\xa4\x85\xb5\x26\xcf\xbc\x16\xd3\
\xfe\x0b\xb8\xfa\xb1\xfb\x34\xf8\x16\x76\x92\x96\xe3\xee\x97\xf1\
\xc1\xad\x15\xe3\x0f\x18\x3a\x5e\x2f\xfe\x14\x1a\xdd\x1b\xe9\x71\
\x11\xc8\xc3\xaa\xf2\xa0\xca\x22\x7d\x91\xcd\xc8\x01\x5f\x38\x48\
\xe9\x6a\xbe\xf4\x1a\x42\xcc\x6d\xa2\x29\x5c\x1d\x66\xb6\x38\x1c'
→  r2con2025
```

Technical Analysis - String Encryption

Result:

```
(venv) ➜  r2con2025 python3 test_data_vmcode.py data.vmcode
File: data.vmcode
Size: 203523 bytes
Key length: 255 bytes
Key first 2 bytes (length marker): 5803

============================================================
Searching for strings (2-byte aligned scan)...
============================================================

0x0000b6: yWUKCt
0x0000ea: hTX
0x00011a: lib/x86/libmapbufferjni.so
0x000152: anwhcacYyM
0x00017a: Delegate logger cannot be null at this state.
0x0001e6: UbEvkSwTN
0x00032c: PASTE
0x000366: BxCUTpJ
0x0003de: VrkRvSB
0x000596: VpNFfxDS
0x0005c8: , musicType=
0x0005f2: ACTION_UNKNOWN
0x000632: hjkuCswiTmFBS
0x0006d8: eVgVE
0x000708: kYPhf
0x000740: profileNavigator
0x000848: FNFbQsw
0x0008a6: Egrqy
0x0008ae: dQNPvpKGyF
0x000906: QbQRjCYL
0x000934: HJpypLvqzlZOea
0x00094c: QYqGLc
0x000954: MnqDC
0x0009d0:  because ACTION_DOWN was not received for this pointer before ACTION_MOVE. It likely happened because  ViewDragHelper did not receive all the events in the event stream.
0x000a88: jLZhKbrVJQpW
0x000b48: eSnc
0x000b4e: OREmOlgHfdyJ
0x000bcc: secondary_button
0x000d6e: Qmx
0x000d90:  deleted
0x000d9a: cross mark
0x000e34: hXkjHDaaacWE
0x000e4a: onDismiss called for DialogFragment
0x000f78: tLiQLU
0x000fb0: IBG-Surveys
0x001044: user0
0x00115e: tLqoBOBARj
0x00118a: uLp
0x0011f6: projects/%s/installations
0x001264: isDouble
0x00126e: 6TJHGK3LCSkGEByU
0x001280: AUzlyL
0x001342: EUHddC
0x00134e: f09f96b1efb88f
0x001362: mGLpxrwL
0x00136c: next_cursor
0x0013ac: MLyIMHQKcouRBC
0x00141e: SjHH
```

Final

Thanks for your attention!


https://github.com/ahmeth4n


https://linkedin.com/in/ahmethangultekin


ahmethan@byterialab.com