Playing with Binary Analysis

Deobfuscation of VM based software protection

•••

Jonathan Salwan,
Sébastien Bardin and Marie-Laure Potet
SSTIC 2017

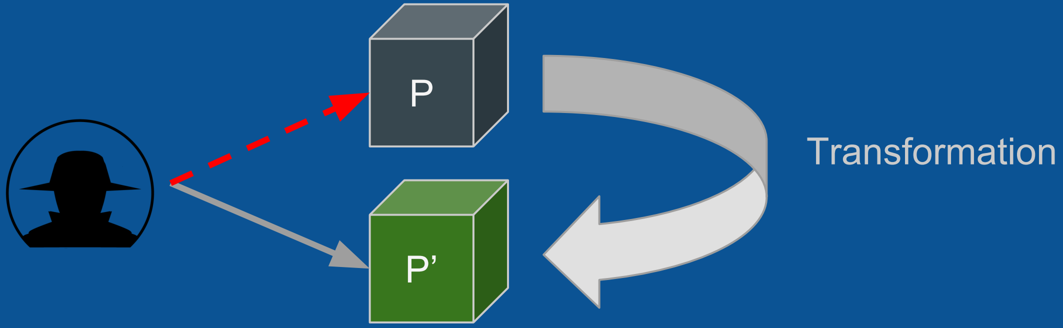
Topic

- Binary protection
    - Virtualization-based software protection
- Automatic deobfuscation, our approach
- The Tigress challenges
- Limitations
- What next?
- Conclusion

Binary Protection

Binary Protection

- Goal
  - Turn your program to make it hard to analyze
    - Protect your software against reverse engineering



Binary Protection

- There are several kinds of protection
  - [...]
  - Virtualization-based software protection

Binary Protection - Virtualization

- Also called Virtual Machine (VM)
- Virtualize a custom Instruction Set Architecture (ISA)

Binary Protection - Virtualization

- Also called Virtual Machine (VM)
- Virtualize a custom Instruction Set Architecture (ISA)

```
long secret(long x) {
        [transformations on x]
        return x;
}


bool auth(long user_input) {
        long h = secret(user_input);
        return (h == 0x9e3779b97f4a7c13);
}
```

Binary Protection - Virtualization

- Also called Virtual Machine (VM)
- Virtualize a custom Instruction Set Architecture (ISA)

```
long secret(long x) {
        [transformations on x]
        return x;
}
```

Bytecodes - Custom ISA

```
bool auth(long user_input) {
        long h = secret(user_input);
        return (h == 0x9e3779b97f4a7c13);
}
```

Binary Protection - Virtualization

- Also called Virtual Machine (VM)
- Virtualize a custom Instruction Set Architecture (ISA)

```
long secret(long x) {
        [transformations on x]
        return x;
}
```

Bytecodes - Custom ISA

```
bool auth(long user_input) {
        long h = 0;
        VM(opcodes, &h, user_input);
        return (h == 0x9e3779b97f4a7c13);
}
```

Binary Protection - Virtualization

- Also called Virtual Machine (VM)
- Virtualize a custom Instruction Set Architecture (ISA)

Removed

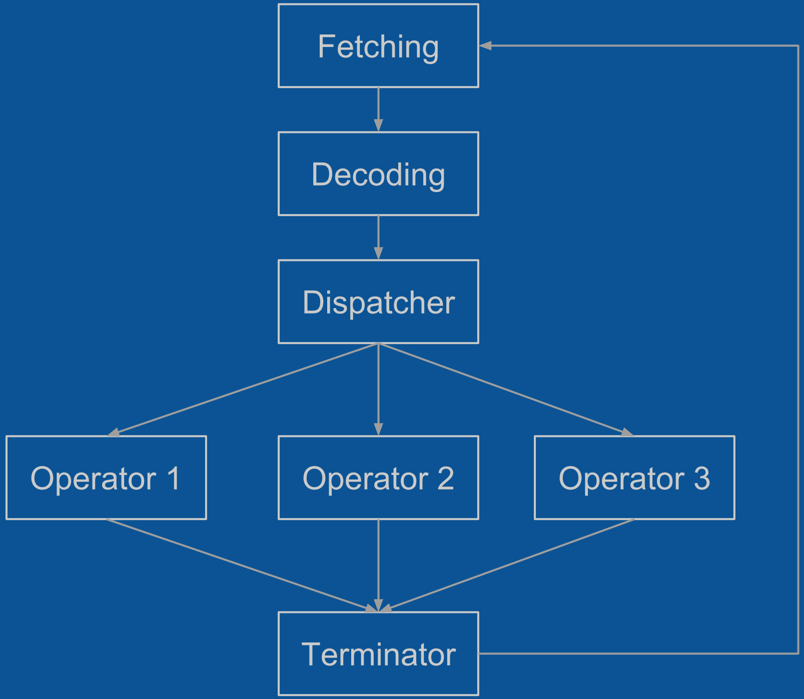
```
long secret(long x) {
        [transformations on x]
        return x;
}
```

Bytecodes - Custom ISA

```
bool auth(long user_input) {
        long h = 0;
        VM(opcodes, &h, user_input);
        return (h == 0x9e3779b97f4a7c13);
}
```

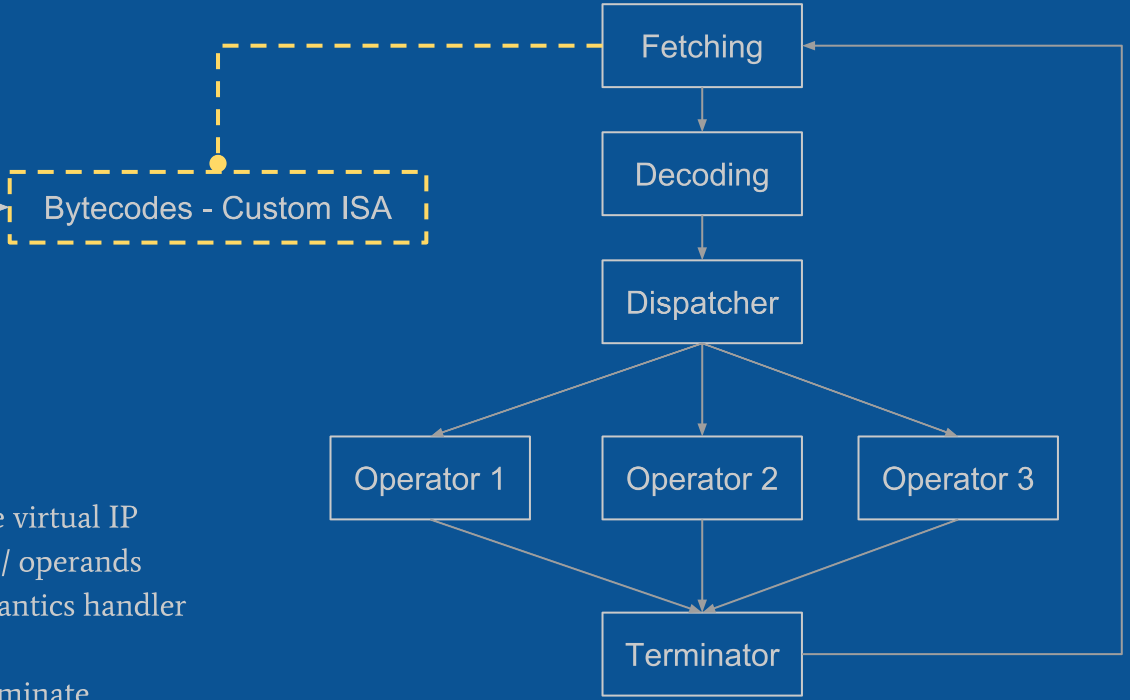
Binary Protection - VM Design (a simple one)

- Close to a CPU design
    a. Fetch the opcode pointed via the virtual IP
    b. Decode the opcode - mnemonic / operands
    c. Dispatch to the appropriate semantics handler
    d. Execute the semantics
    e. Go to the next instruction or terminate



Binary Protection - VM Design (a simple one)

```
long secret(long x) {
        [transformations on x]
        return x;
}
```

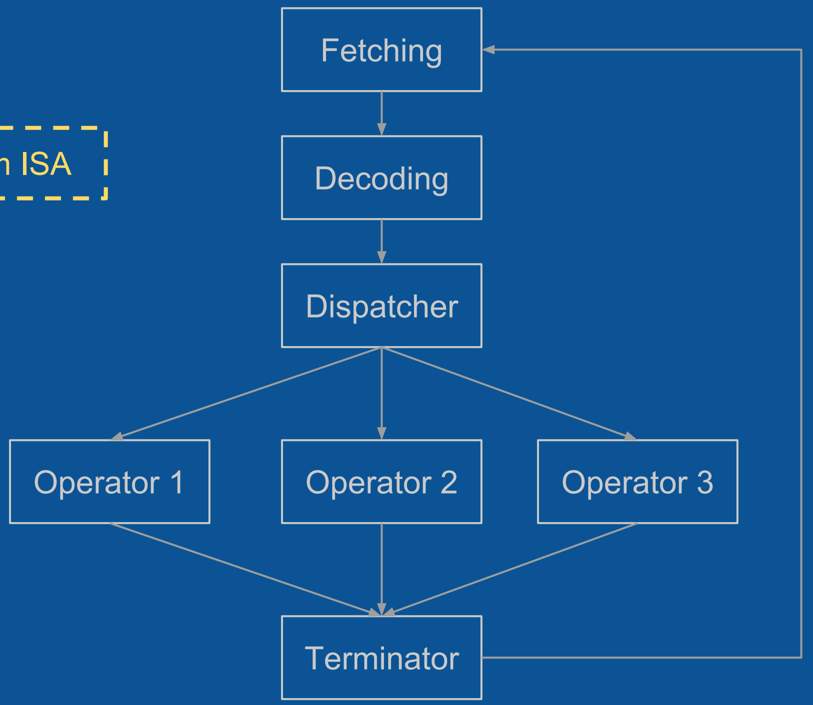


- Close to a CPU design
  a. Fetch the opcode pointed via the virtual IP
  b. Decode the opcode - mnemonic / operands
  c. Dispatch to the appropriate semantics handler
  d. Execute the semantics
  e. Go to the next instruction or terminate

Binary Protection - VM Design (a simple one)

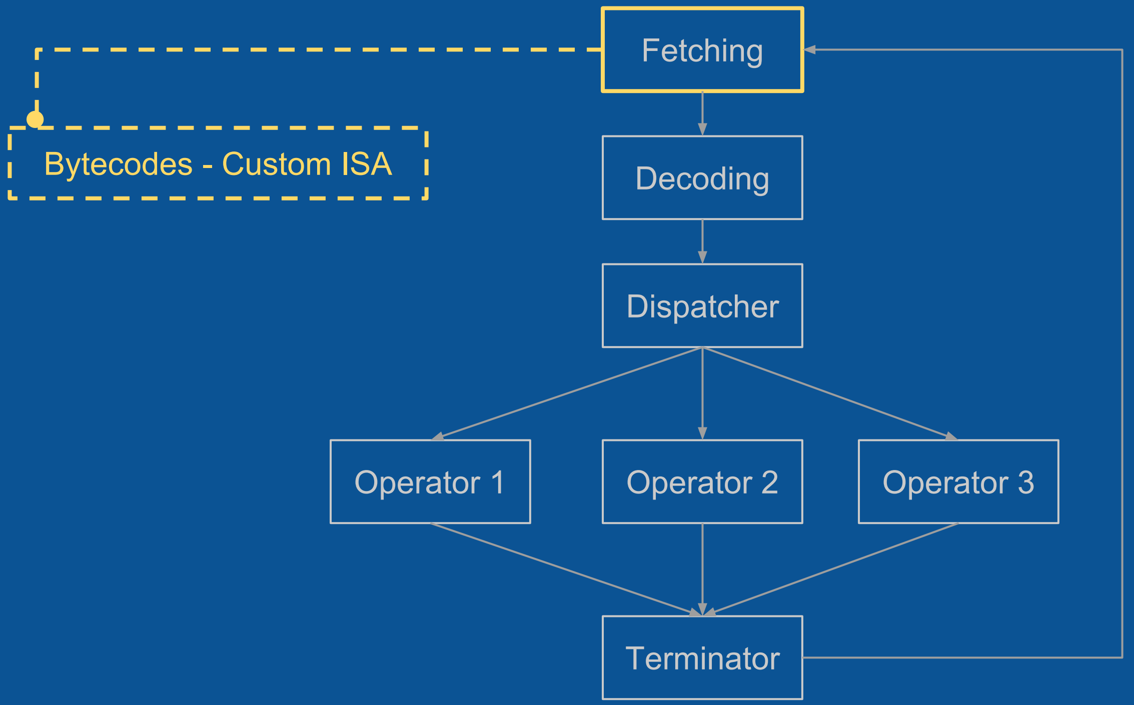
Bytecodes - Custom ISA

```
Fetch  :
Decode :

Code   :
```



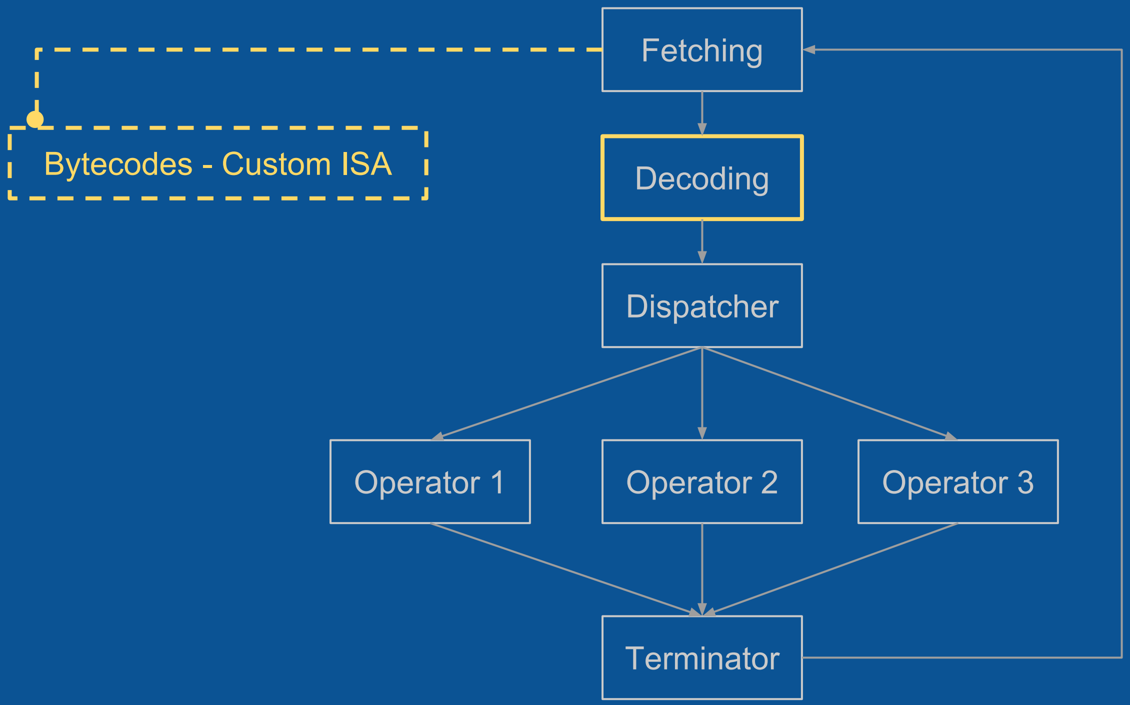
Binary Protection - VM Design (a simple one)

```
Fetch  : 0xaabbccdd
Decode :

Code   :
```

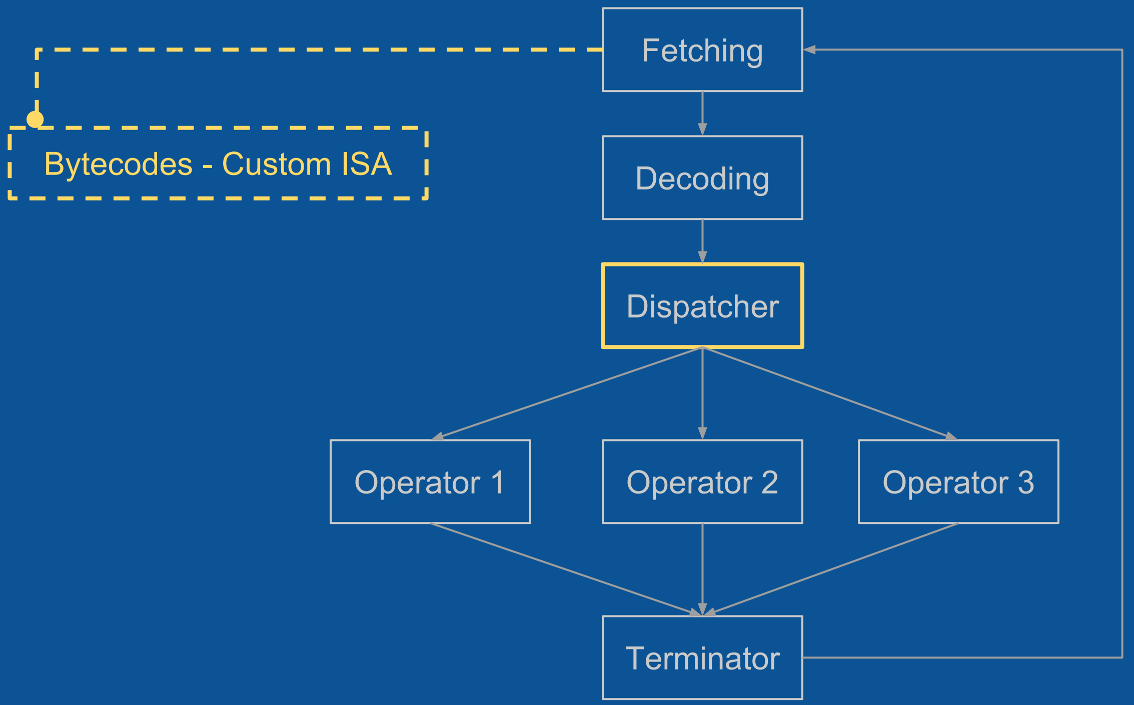


Binary Protection - VM Design (a simple one)

```
Fetch  : 0xaabbccdd
Decode : mov r/r

Code   :
```

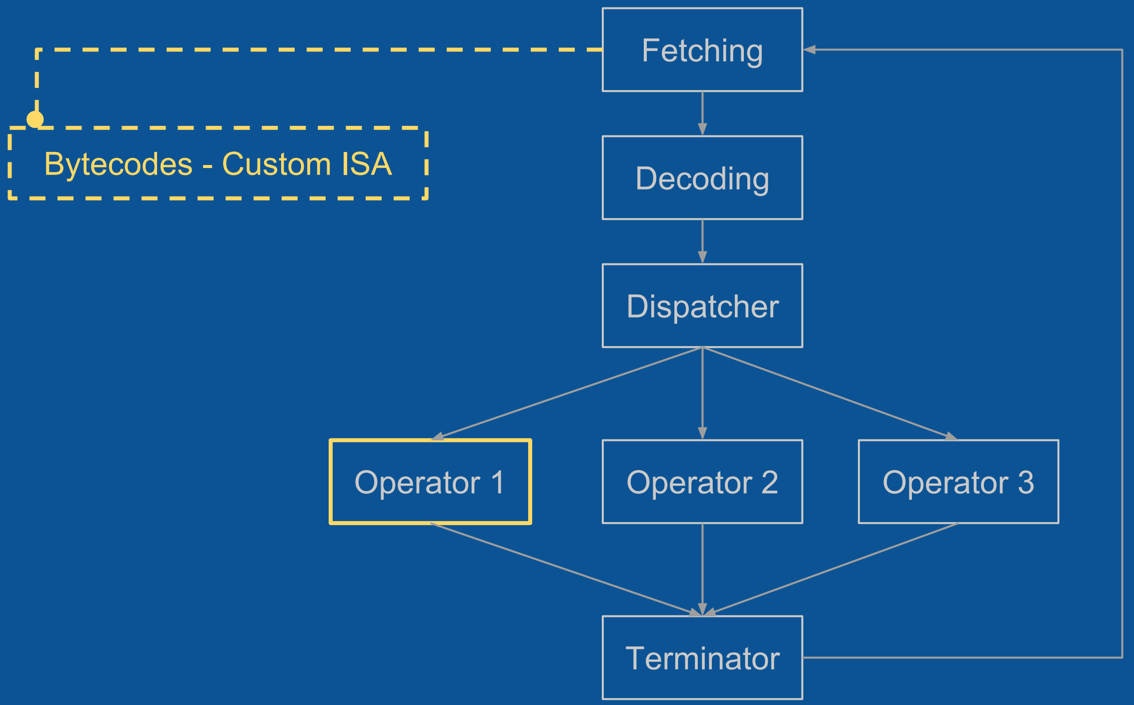


Binary Protection - VM Design (a simple one)




```
Fetch  : 0xaabbccdd
Decode : mov r/r

Code   :
```

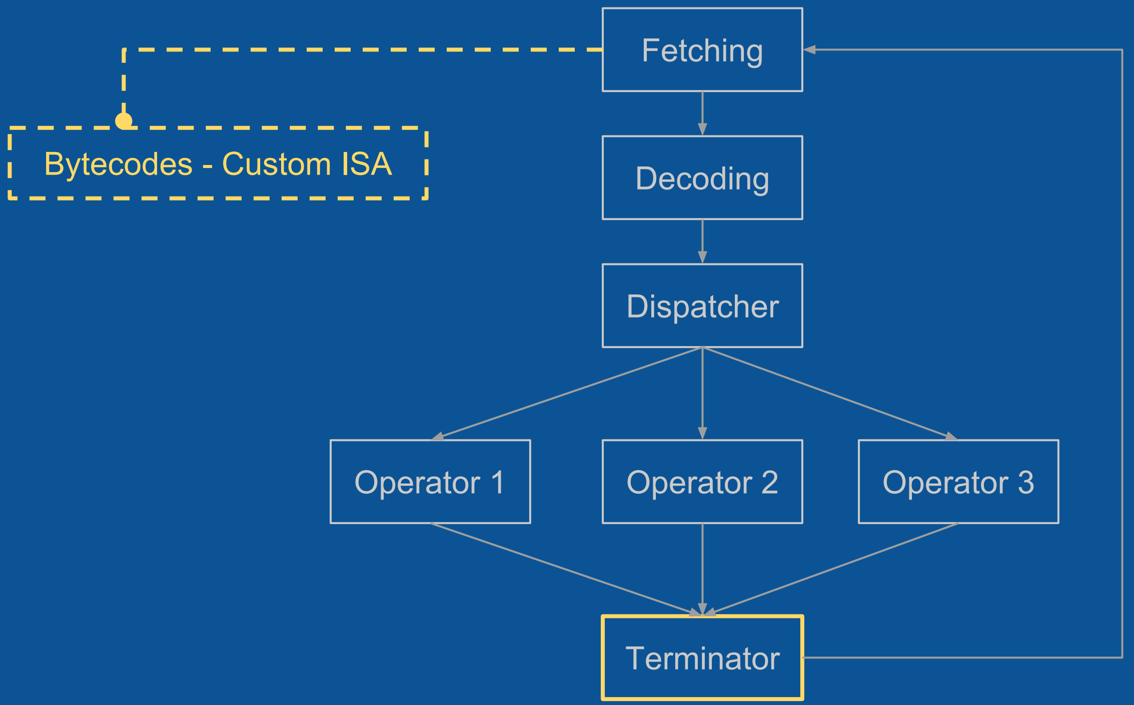
Binary Protection - VM Design (a simple one)




```
Fetch   : 0xaabbccdd
Decode  : mov r/r

Code    : mov r1, input
```

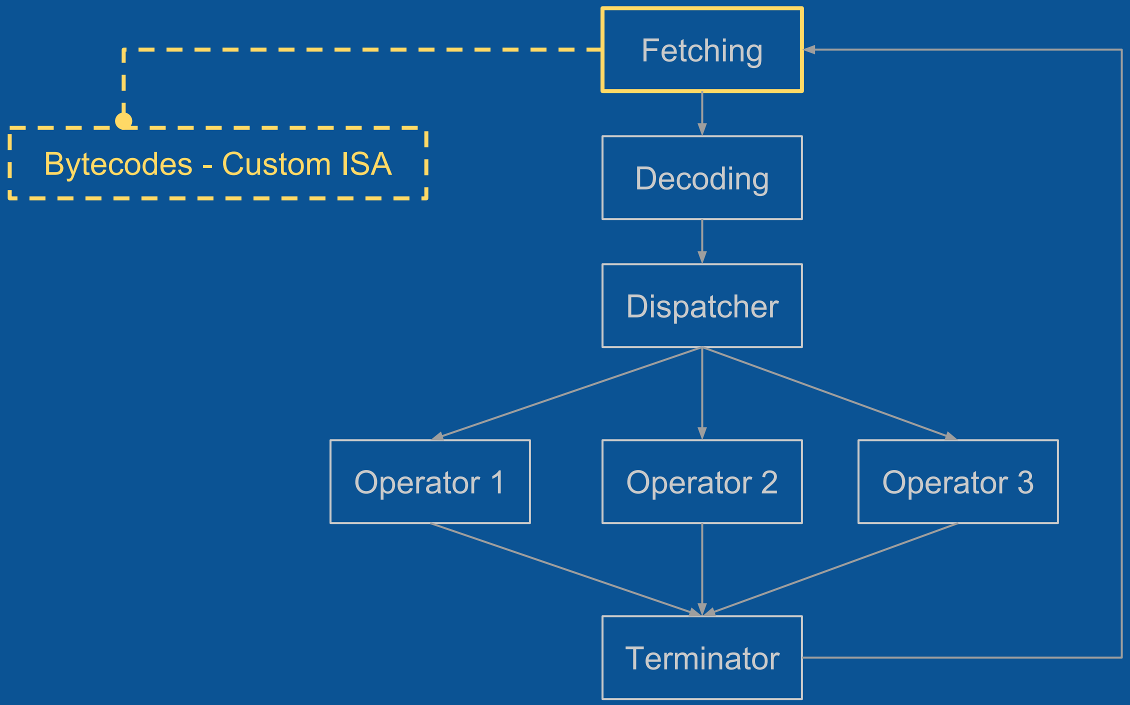
Binary Protection - VM Design (a simple one)

```
Fetch  :
Decode :

Code   : mov r1, input
```



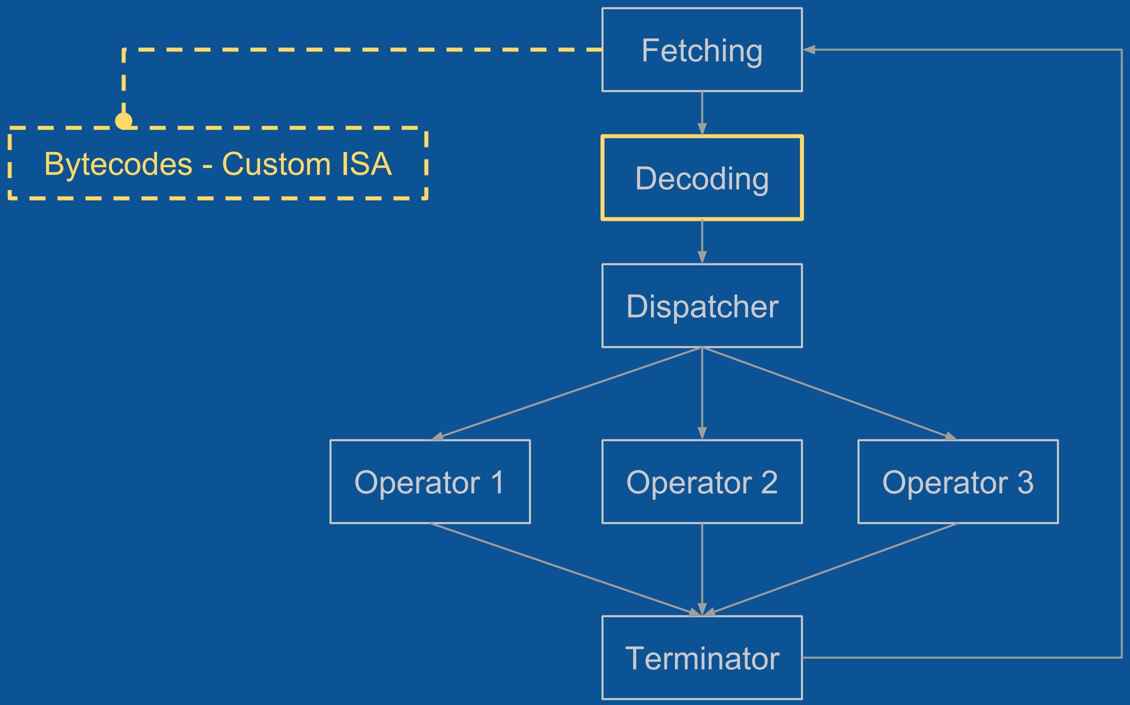
Binary Protection - VM Design (a simple one)

```
Fetch  : 0x11223344
Decode :

Code   : mov r1, input
```



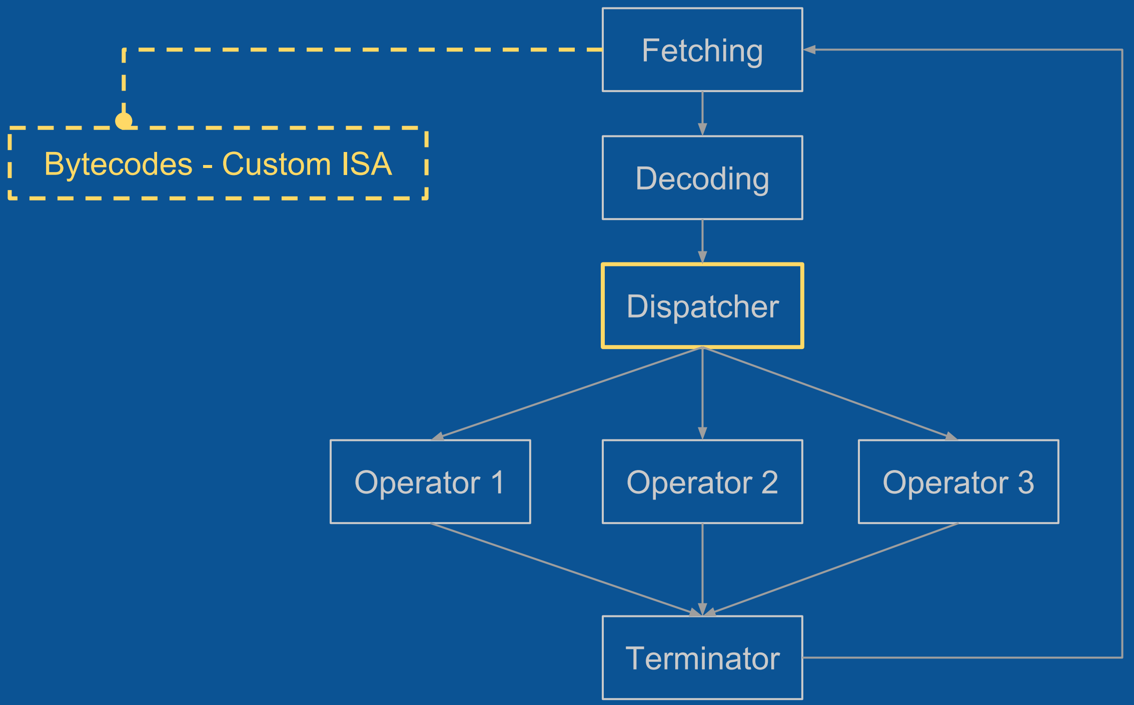
Binary Protection - VM Design (a simple one)

```
Fetch  : 0x11223344
Decode : mov r/i

Code   : mov r1, input
```



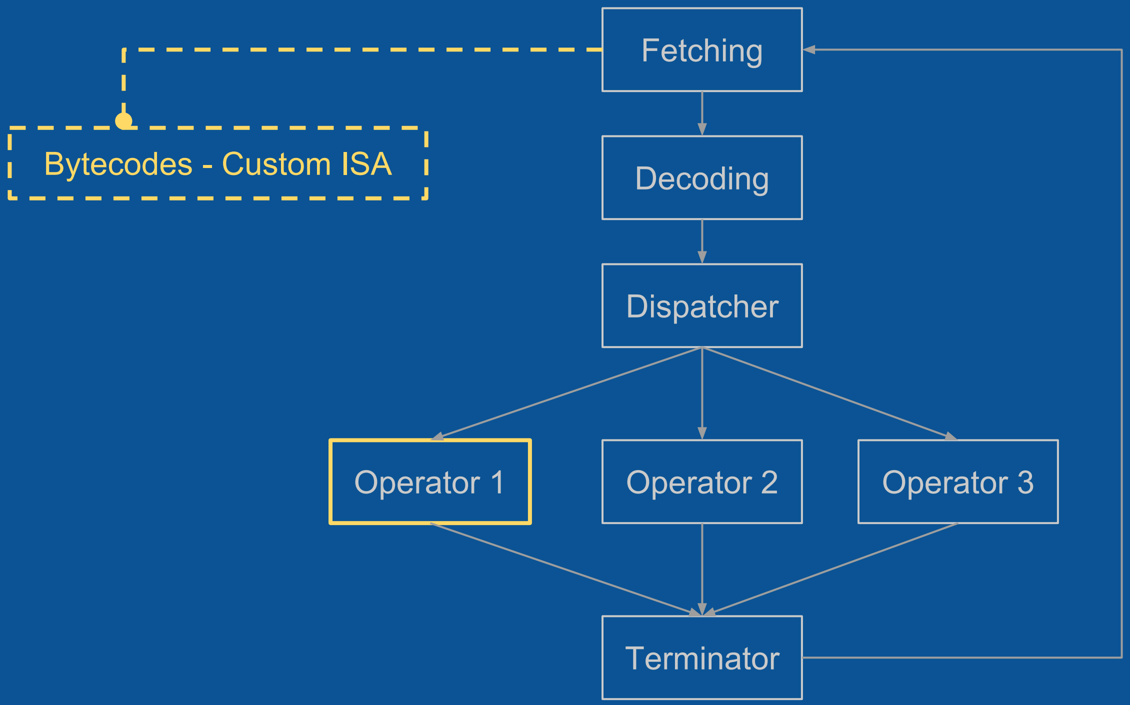
Binary Protection - VM Design (a simple one)

```
Fetch   : 0x11223344
Decode  : mov r/i

Code    : mov r1, input
```

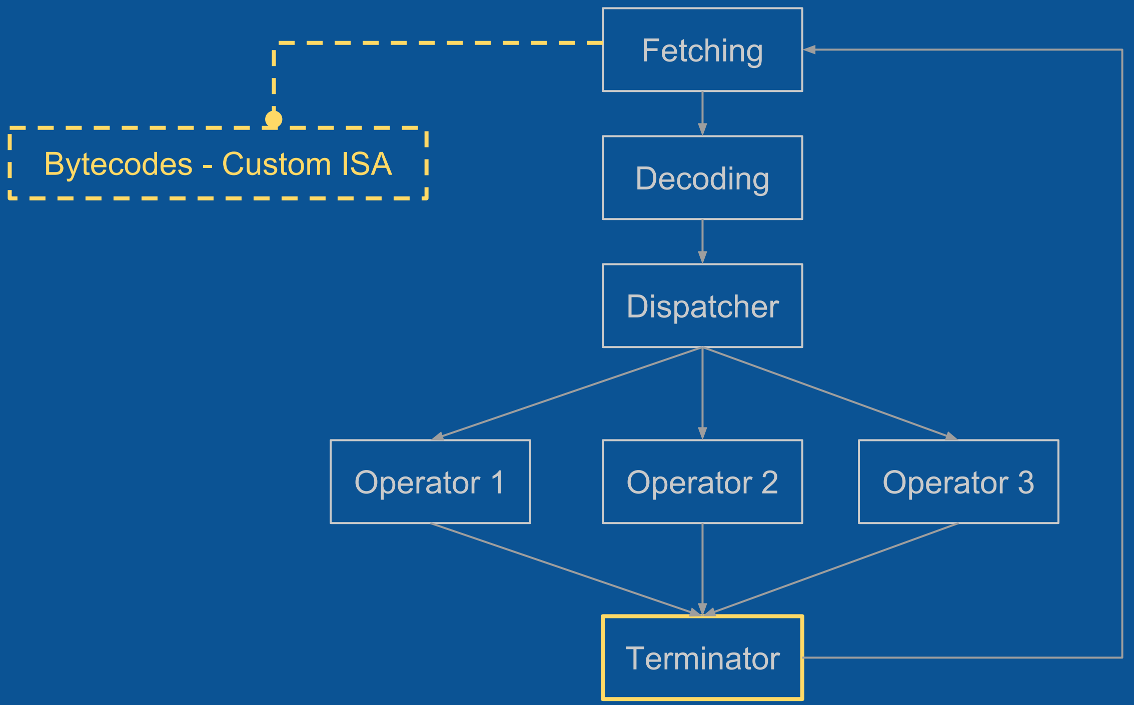


Binary Protection - VM Design (a simple one)

```
Fetch   : 0x11223344
Decode  : mov r/i

Code    : mov r1, input
          mov r2, 2
```



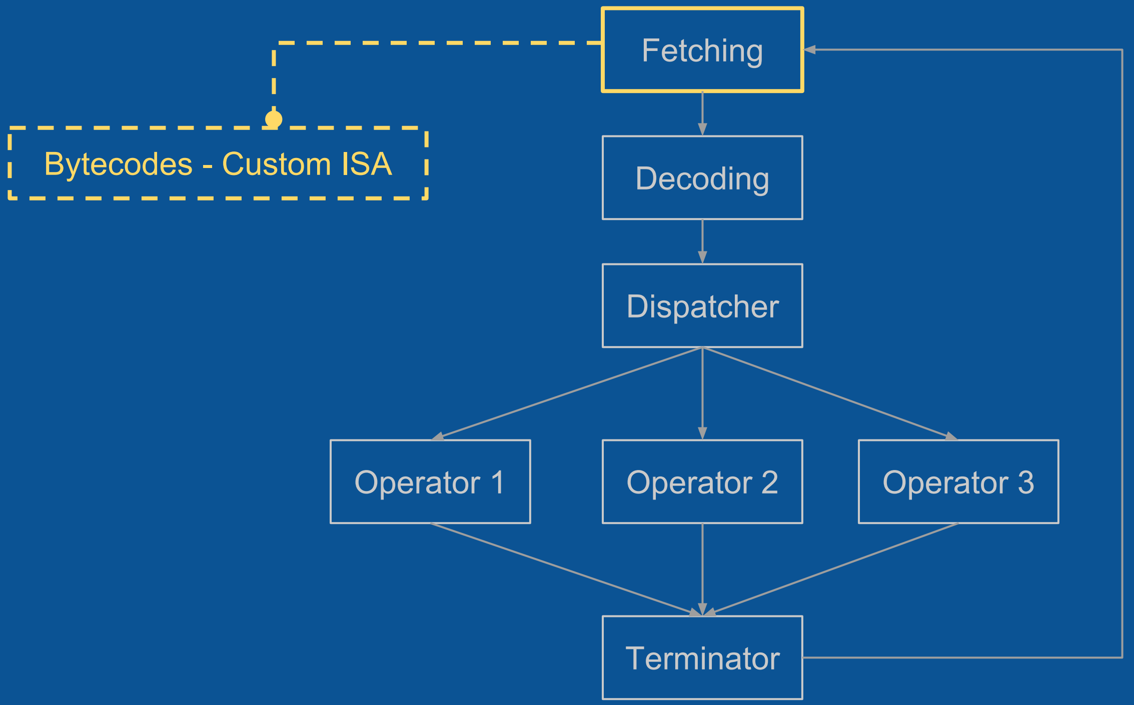
Binary Protection - VM Design (a simple one)



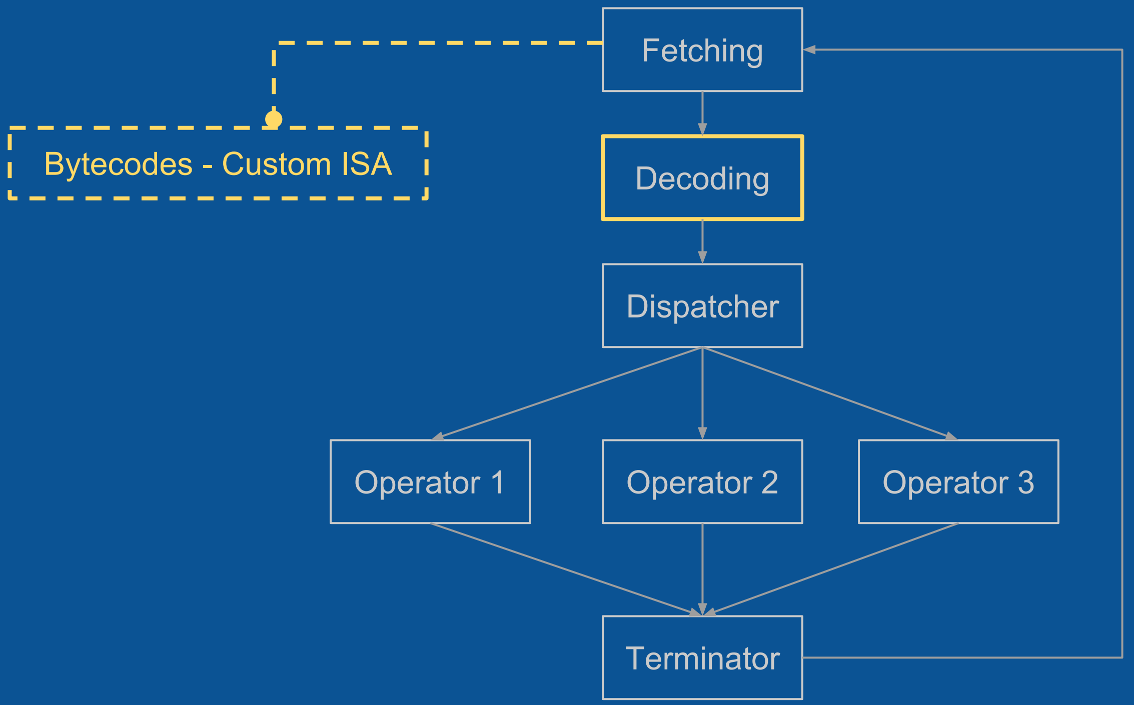

```
Fetch   :
Decode  :

Code    : mov r1, input
          mov r2, 2
```

Binary Protection - VM Design (a simple one)

```
Fetch   : 0x5577aabb
Decode  :

Code    : mov r1, input
          mov r2, 2
```



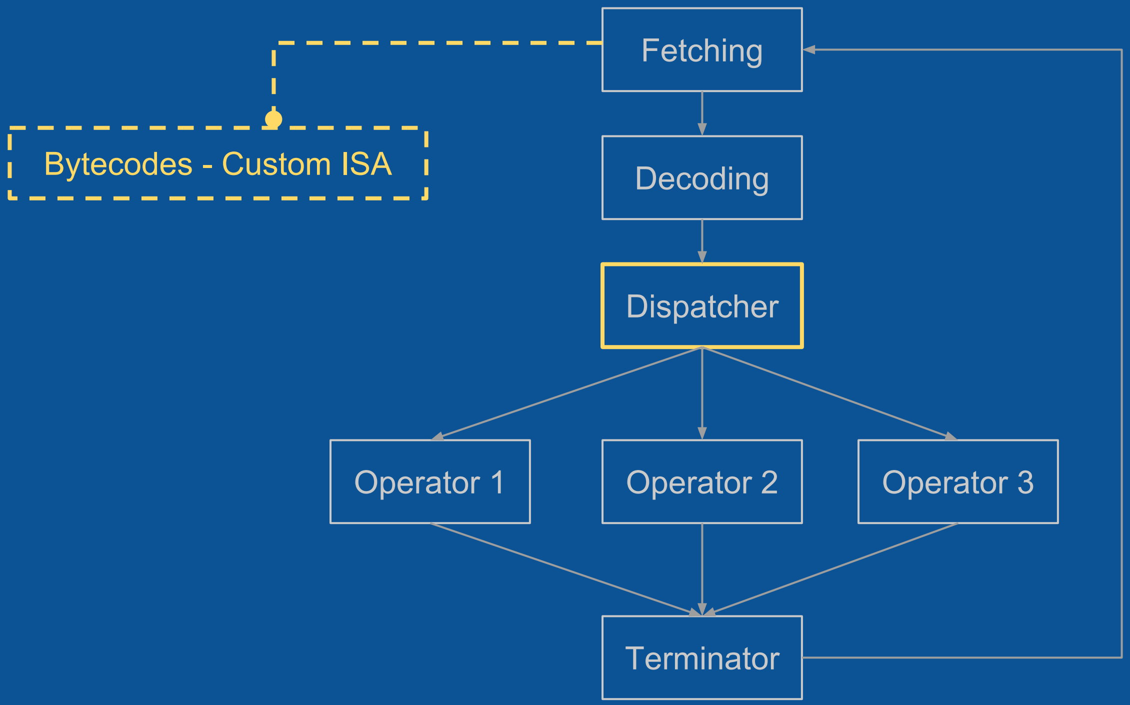
Binary Protection - VM Design (a simple one)




```
Fetch  : 0x5577aabb
Decode : mul r/r/r

Code   : mov r1, input
         mov r2, 2
```

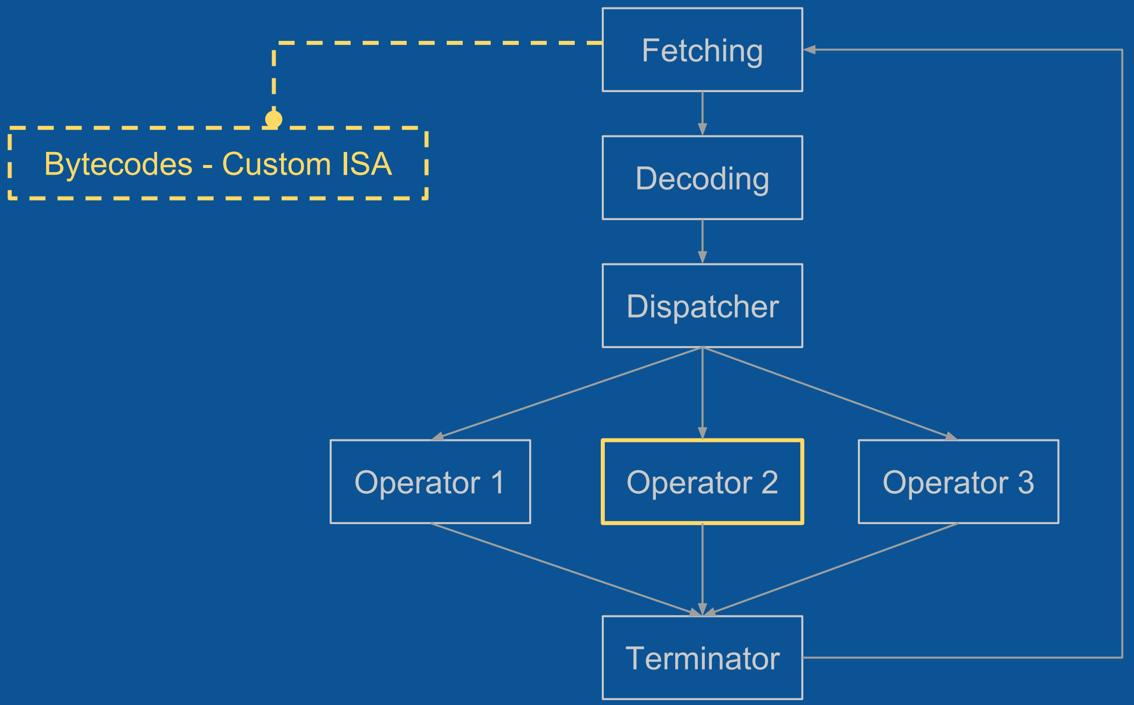
Binary Protection - VM Design (a simple one)

```
Fetch  : 0x5577aabb
Decode : mul r/r/r

Code   : mov r1, input
         mov r2, 2
```



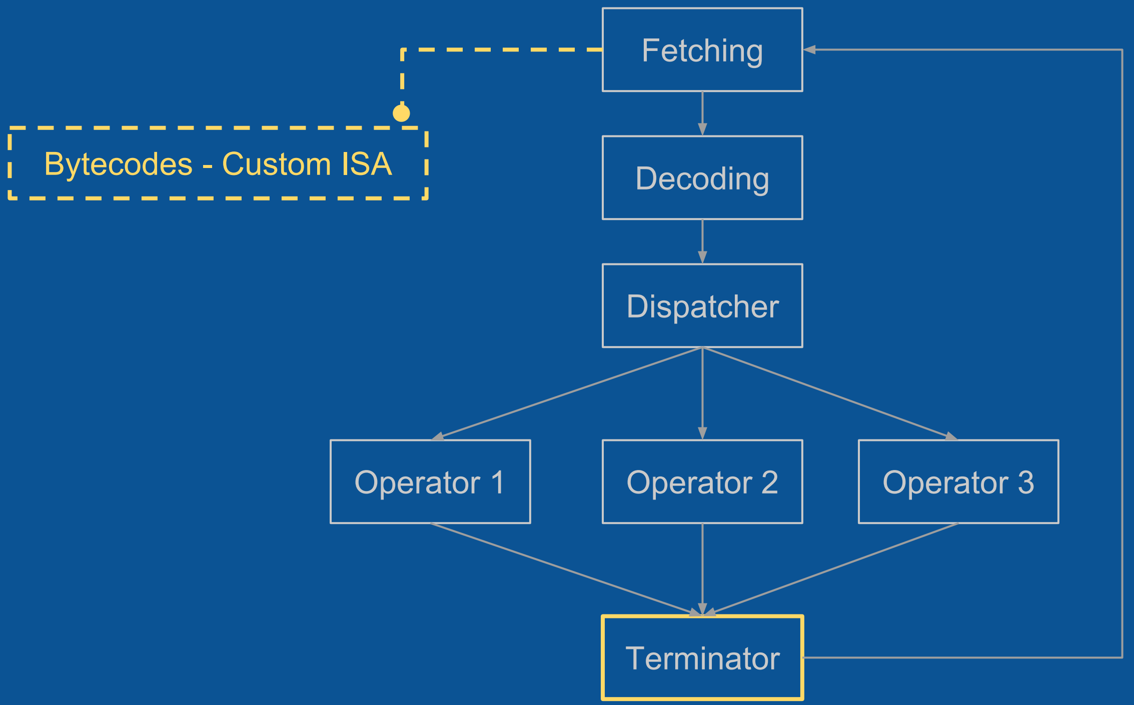
Binary Protection - VM Design (a simple one)

```
Fetch   : 0x5577aabb
Decode  : mul r/r/r

Code    : mov r1, input
          mov r2, 2
          mul r3, r1, r2
```



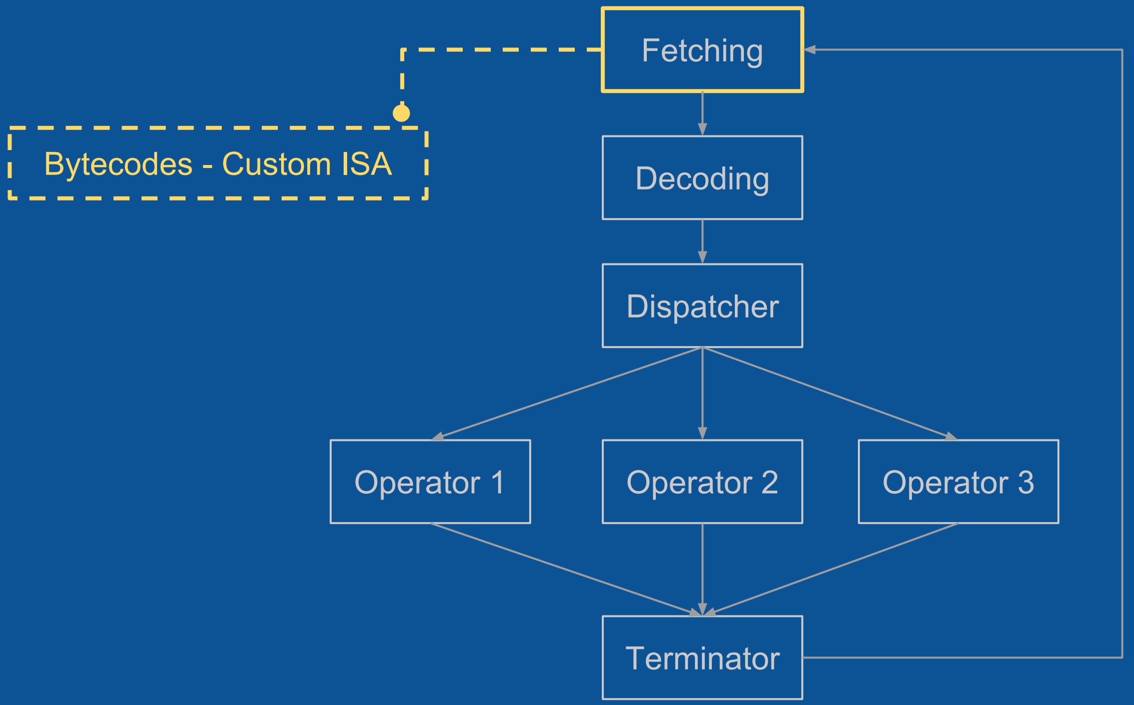
Binary Protection - VM Design (a simple one)

```
Fetch   :
Decode  :

Code    : mov r1, input
          mov r2, 2
          mul r3, r1, r2
```



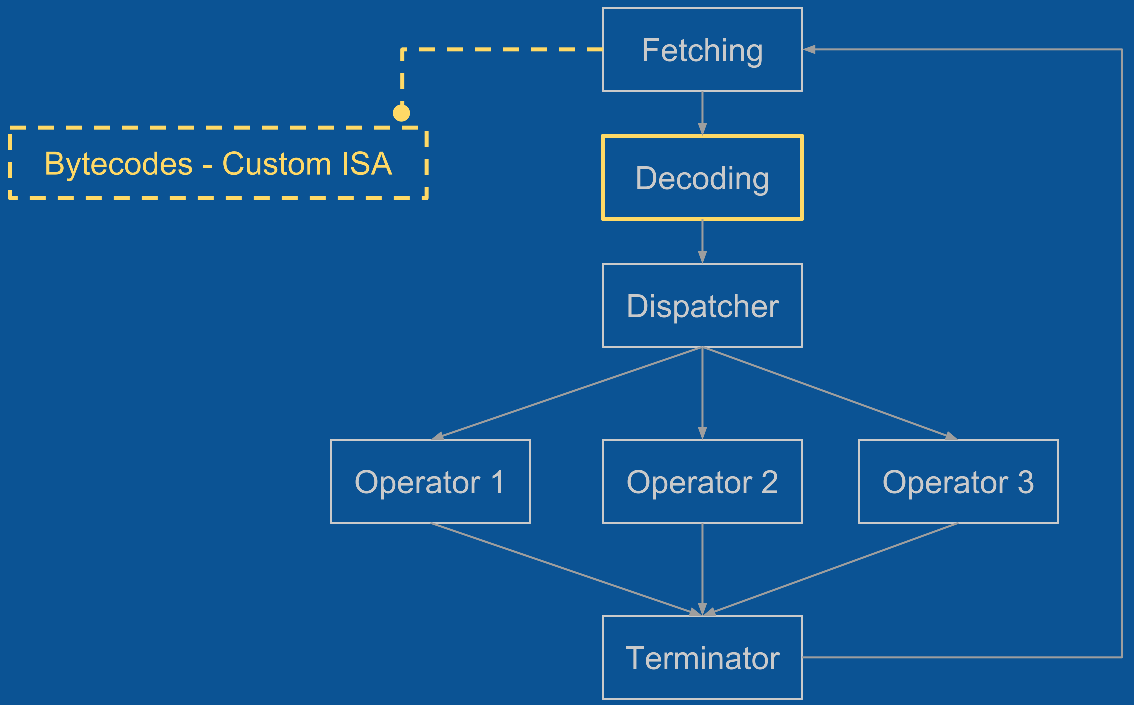
Binary Protection - VM Design (a simple one)

```
Fetch   : 0x1337dead
Decode  :

Code    : mov r1, input
          mov r2, 2
          mul r3, r1, r2
```



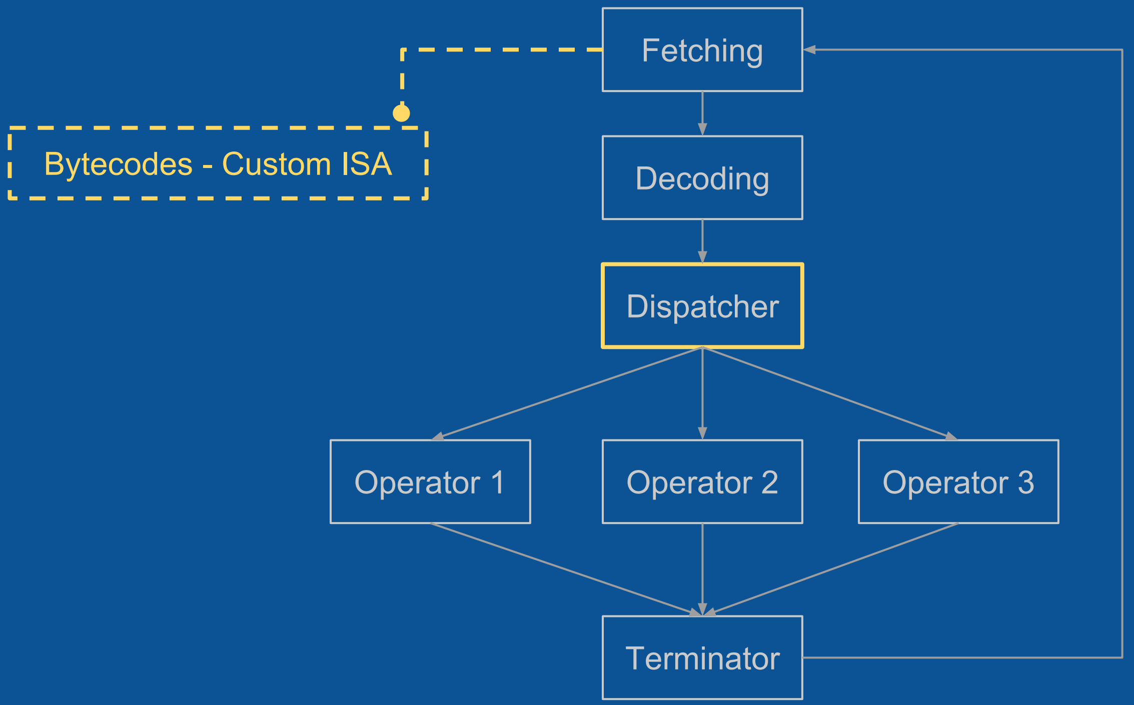
Binary Protection - VM Design (a simple one)

```
Fetch   : 0x1337dead
Decode  : ret r

Code    : mov r1, input
          mov r2, 2
          mul r3, r1, r2
```



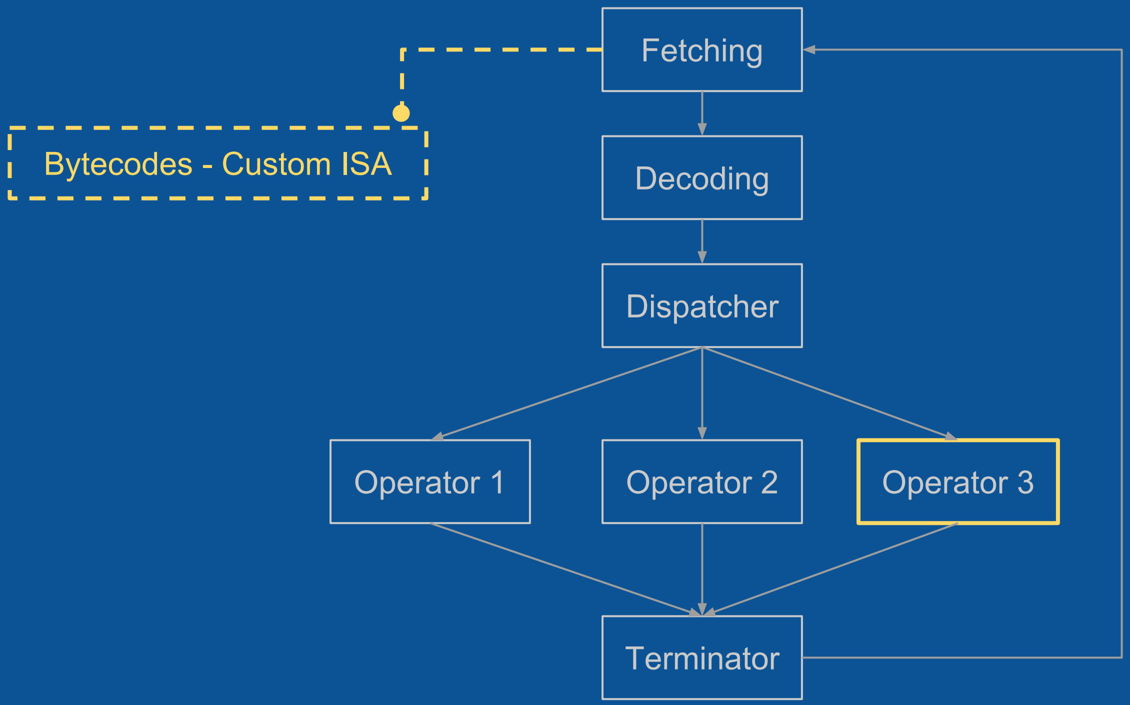
Binary Protection - VM Design (a simple one)

```
Fetch  : 0x1337dead
Decode : ret r

Code   : mov r1, input
         mov r2, 2
         mul r3, r1, r2
```



Binary Protection - VM Design (a simple one)

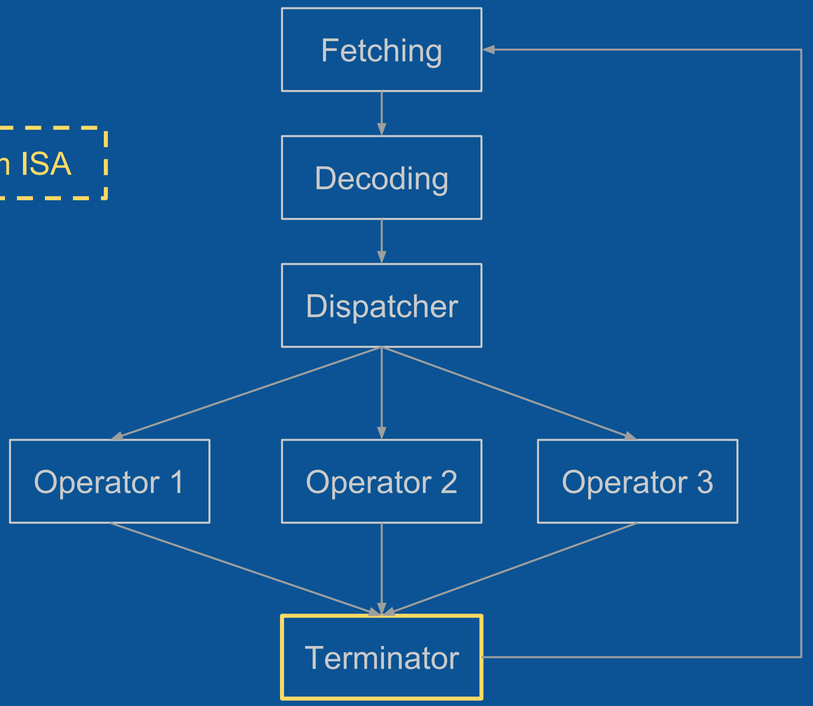
```
Fetch  : 0x1337dead
Decode : ret r

Code   : mov r1, input
         mov r2, 2
         mul r3, r1, r2
         ret r3
```



Binary Protection - VM Design (a simple one)

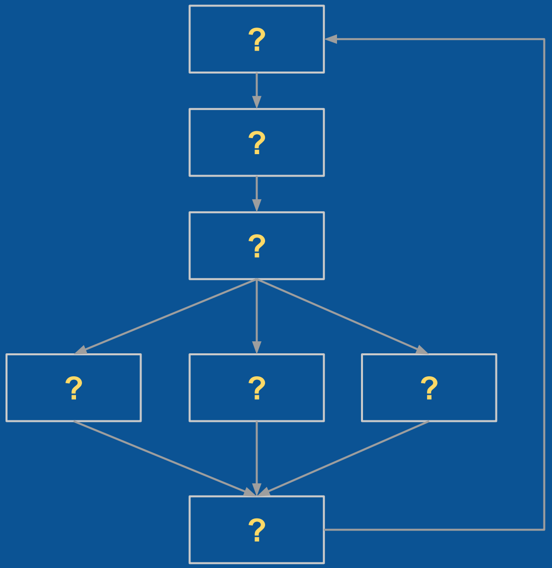
Bytecodes - Custom ISA

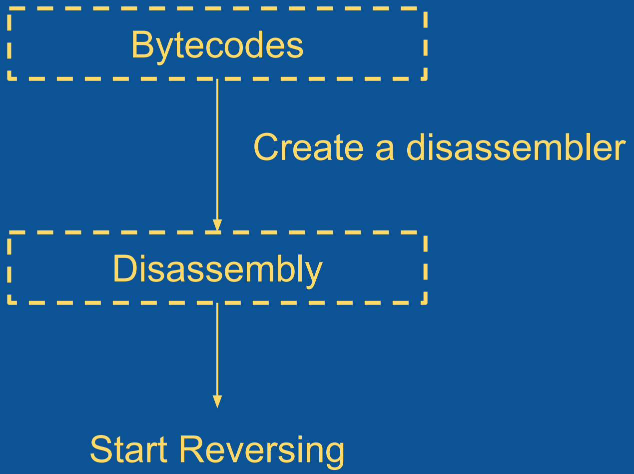
```
Fetch  :
Decode :

Code   : mov r1, input
         mov r2, 2
         mul r3, r1, r2
         ret r3
```



Virtual Machine - Standard Reverse Process

- Reverse and understand the virtual machine's structure / components
- Create a disassembler and then reverse the bytecodes





Our Approach
Automatic Deobfuscation

Our Approach - Automatic Deobfuscation

- We don’t care about reconstructing a disassembler
- Our goal:

Our Approach - Automatic Deobfuscation

- We don’t care about reconstructing a disassembler
- Our goal:
  - Directly reconstruct a devirtualized binary from the obfuscated one

Our Approach - Automatic Deobfuscation

- We don’t care about reconstructing a disassembler
- Our goal:
  - Directly reconstruct a devirtualized binary from the obfuscated one
  - The crafted binary must have a control flow graph close to the original one

Our Approach - Automatic Deobfuscation

- We don’t care about reconstructing a disassembler
- Our goal:
    - Directly reconstruct a devirtualized binary from the obfuscated one
    - The crafted binary must have a control flow graph close to the original one
    - The crafted binary must have instructions close to the original ones

Our Approach - Automatic Deobfuscation

Removed

```
long secret(long x) {
        [transformations on x]
        return x;
}
```

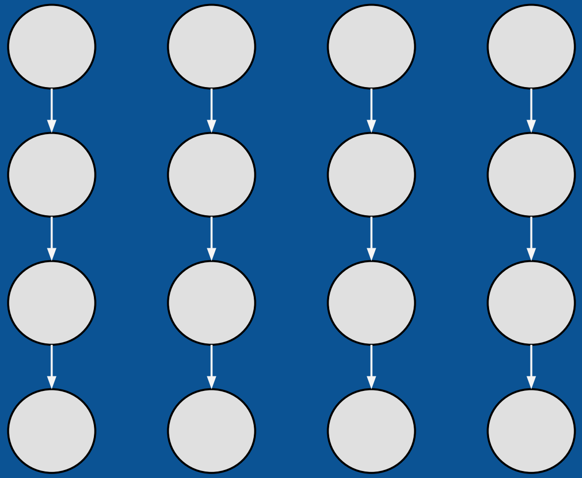
Bytecodes

FROM

```
bool auth(long user_input) {
        long h = 0;
        VM(opcodes, &h, user_input);
        return (h == 0x9e3779b97f4a7c13);
}
```

Our Approach - Automatic Deobfuscation

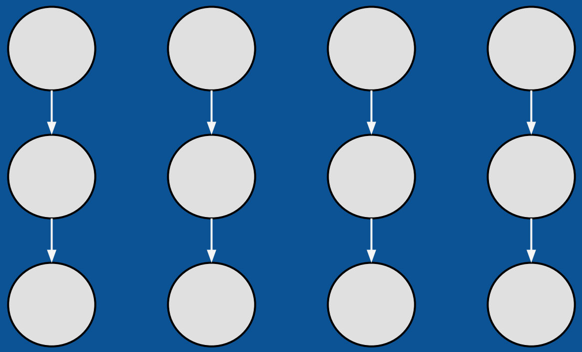
TO



Obfuscated Traces

Our Approach - Automatic Deobfuscation

THEN FROM



Simplified Traces

Our Approach - Automatic Deobfuscation

```
long secret_prime(long x) {
        [transformations on x]
        return x;
}
```

TO

```
bool auth(long user_input) {
        long h = secret(user_input);
        return (h == 0x9e3779b97f4a7c13);
}
```

Our Approach - Automatic Deobfuscation

TO

```
long secret_prime(long x) {
        [transformations on x]
        return x;
}
```

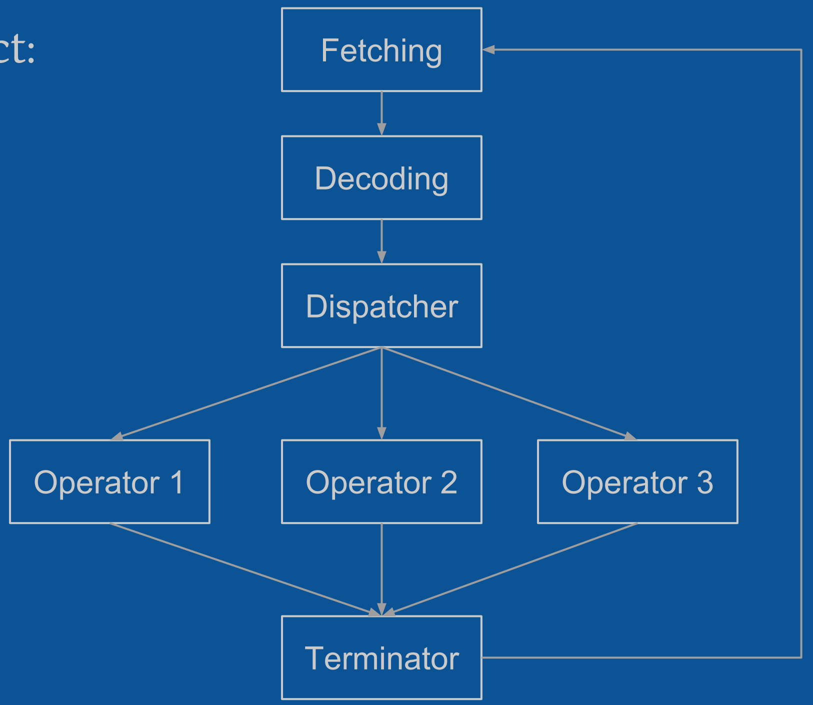
Where *secret_prime()* is semantically identical to the original code but without the process of the virtual machine

```
bool auth(long user_input) {
        long h = secret(user_input);
        return (h == 0x9e3779b97f4a7c13);
}
```

Our Approach - Important fact

- Our approach is based on an important fact:
  - trace P' = instr P + instr VM

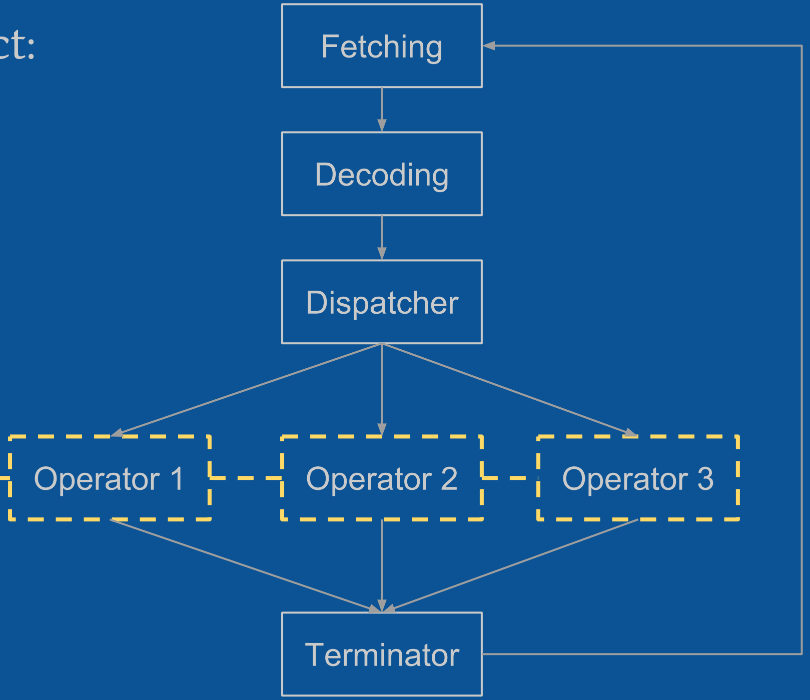
Whatever the process of the VM execution, at the end, it must execute the original instruction (or its equivalent, e.g: div / shr)



Our Approach - Important fact

- Our approach is based on an important fact:
  - trace P' = instr P + instr VM

Whatever the process of the VM execution, at the end, it must execute the original instruction (or its equivalent, e.g: div / shr)



Our Approach - Overview

1. Isolate these pertinent instructions using a taint analysis along a trace
2. Keep a semantics transition between these isolated instructions using a SE
3. Concretize everything which is not related to these instructions (discard VM)
4. Perform a code coverage to recover the original CFG (iterate on more traces)
5. Transform our representation into the LLVM one
    a. Unfolding program (tree-like program)
6. Recompile with compiler optimizations
    a. Compacted program (folding program)

Step 1: Taint Analysis

- Track the input(s) of the function into the process of the VM execution

```
long secret(long x) {
        [transformations on x]
        return x;
}
```

Custom ISA

```
bool auth(long user_input) {
        long h = 0;
        VM(opcodes, &h, user_input);
        return (h == 0x9e3779b97f4a7c13);
}
```

Tainted

Step 1: Taint Analysis

- Track the input(s) of the function into the process of the VM execution
- Pertinent instructions isolated

```
mov   rsi, qword ptr [rax]
mov   rbx, rsi
shr   rbx, cl
mov   rax, rbx
mov   qword ptr [rdx], rax

mov   rdx, qword ptr [rdx]
mov   qword ptr [rax], rdx

mov   rcx, qword ptr [rax]
xor   rax, rcx
mov   qword ptr [rdx], rax

[...]
```

```
long secret(long x) {
        [transformations on x]
        return x;
}
```

Custom ISA

```
bool auth(long user_input) {
        long h = 0;
        VM(opcodes, &h, user_input);
        return (h == 0x9e3779b97f4a7c13);
}
```

Tainted

Step 1: Taint Analysis

- Track the input(s) of the function into the process of the VM execution
- Pertinent instructions isolated

```
mov    rsi, qword ptr [rax]
mov    rbx, rsi
shr    rbx, cl
mov    rax, rbx
mov    qword ptr [rdx], rax

mov    rdx, qword ptr [rdx]
mov    qword ptr [rax], rdx

mov    rcx, qword ptr [rax]
xor    rax, rcx
mov    qword ptr [rdx], rax

[...]
```

```
0000000000400838
0000000000400838 loc_400838:                    ; jumptable 000000000040072E case 4
0000000000400838 mov     rax, [rbp+vip]
000000000040083C add     rax, 1
0000000000400840 mov     [rbp+vip], rax
0000000000400844 mov     rax, [rbp+var_80]
0000000000400848 lea     rdx, [rax-8]
000000000040084C mov     rax, [rbp+var_80]
0000000000400850 sub     rax, 8
0000000000400854 mov     rcx, [rax]
0000000000400857 mov     rax, [rbp+var_80]
000000000040085B mov     rax, [rax]
000000000040085E xor     rax, rcx
0000000000400861 mov     [rdx], rax
0000000000400864 mov     rax, [rbp+var_80]
0000000000400868 sub     rax, 8
000000000040086C mov     [rbp+var_80], rax
0000000000400870 jmp     loc_400E6B
```

Step 1: Taint Analysis

- Track the input(s) of the function into the process of the VM execution
- Pertinent instructions isolated
  - Now, the problem is that this sub-trace has no sense without the VM's state

```
mov   rsi, qword ptr [rax]
mov   rbx, rsi
shr   rbx, cl
mov   rax, rbx
mov   qword ptr [rdx], rax

mov   rdx, qword ptr [rdx]
mov   qword ptr [rax], rdx

mov   rcx, qword ptr [rax]
xor   rax, rcx
mov   qword ptr [rdx], rax

[...]
```

```
0000000000400838
0000000000400838 loc_400838:                  ; jumptable 000000000040072E case 4
0000000000400838 mov     rax, [rbp+vip]
000000000040083C add     rax, 1
0000000000400840 mov     [rbp+vip], rax
0000000000400844 mov     rax, [rbp+var_80]
0000000000400848 lea     rdx, [rax-8]
000000000040084C mov     rax, [rbp+var_80]
0000000000400850 sub     rax, 8
0000000000400854 mov     rcx, [rax]
0000000000400857 mov     rax, [rbp+var_80]
000000000040085B mov     rax, [rax]
000000000040085E xor     rax, rcx
0000000000400861 mov     [rdx], rax
0000000000400864 mov     rax, [rbp+var_80]
0000000000400868 sub     rax, 8
000000000040086C mov     [rbp+var_80], rax
0000000000400870 jmp     loc_400E6B
```

Step 2: Symbolic Representation

- A symbolic representation is used to provide a sense to these tainted instructions

```
mov   rsi, qword ptr [rax]
mov   rbx, rsi
shr   rbx, cl
mov   rax, rbx
mov   qword ptr [rdx], rax

mov   rdx, qword ptr [rdx]
mov   qword ptr [rax], rdx

mov   rcx, qword ptr [rax]
xor   rax, rcx
mov   qword ptr [rdx], rax

[...]
```

Step 2: Symbolic Representation

- A symbolic representation is used to provide a sense to these tainted instructions

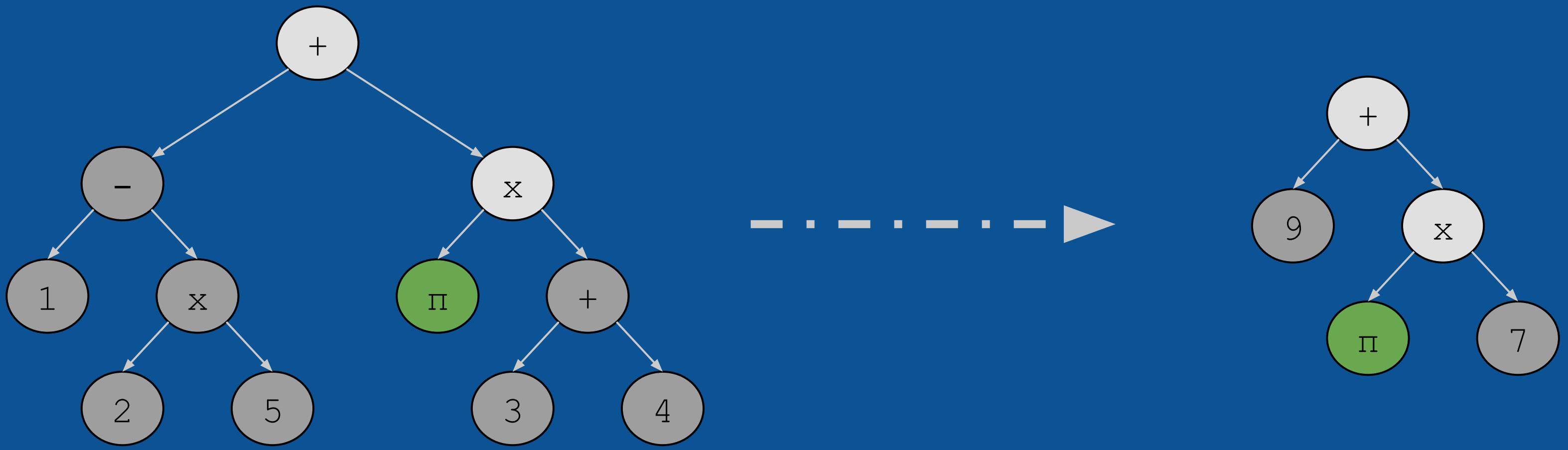
```
mov   rsi, qword ptr [rax]
mov   rbx, rsi
shr   rbx, cl
mov   rax, rbx
mov   qword ptr [rdx], rax

mov   rdx, qword ptr [rdx]
mov   qword ptr [rax], rdx

mov   rcx, qword ptr [rax]
xor   rax, rcx
mov   qword ptr [rdx], rax

[...]
```

Symbolic representation
of a given path

```
ref!228  := SymVar_0
ref!243  := (((_ extract 63 0) ref!228))
ref!1131 := (
  (bvlshr
    ((_ extract 63 0) ref!243)
    (bvand
      ((_ zero_extend 56) (_ bv5 8))
      (_ bv63 64)
    )
  )
)
ref!1334 := (((_ extract 63 0) ref!1131))

[...]
```

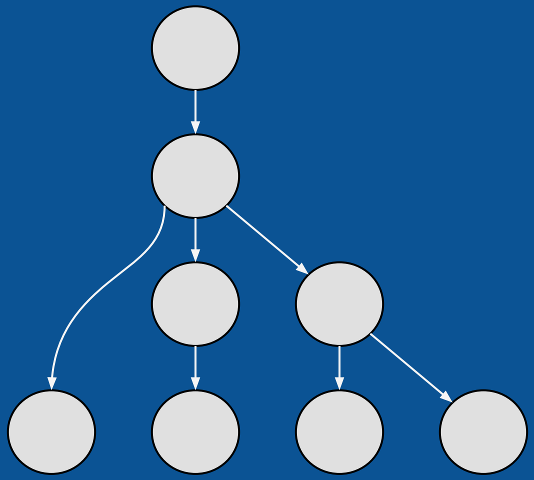
Step 3: Concretization Policy

- Input(s) of the function are both tainted and symbolized
- In order to remove the process of the VM execution
  - We concretize every LOAD and STORE
  - We concretize everything which is not related to the input(s)
    - Untainted values are concretized



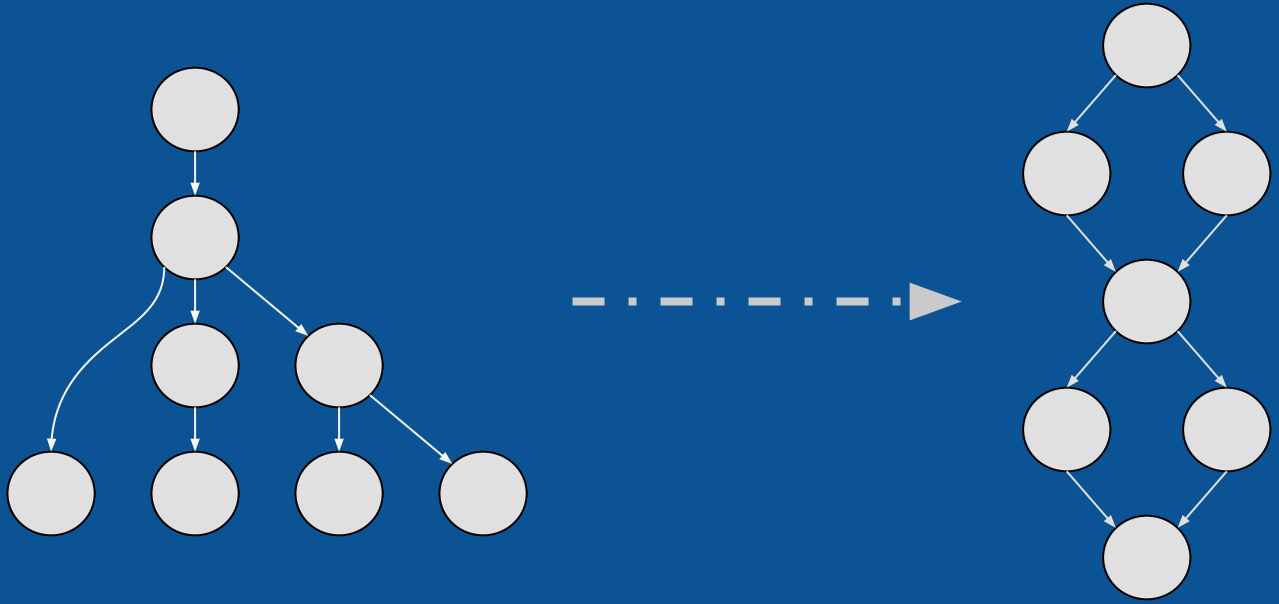
Step 4: Code Coverage - Discovering Paths

- In order to find the original CFG, we must discover its paths
  - SMT solver is used onto our symbolic representation



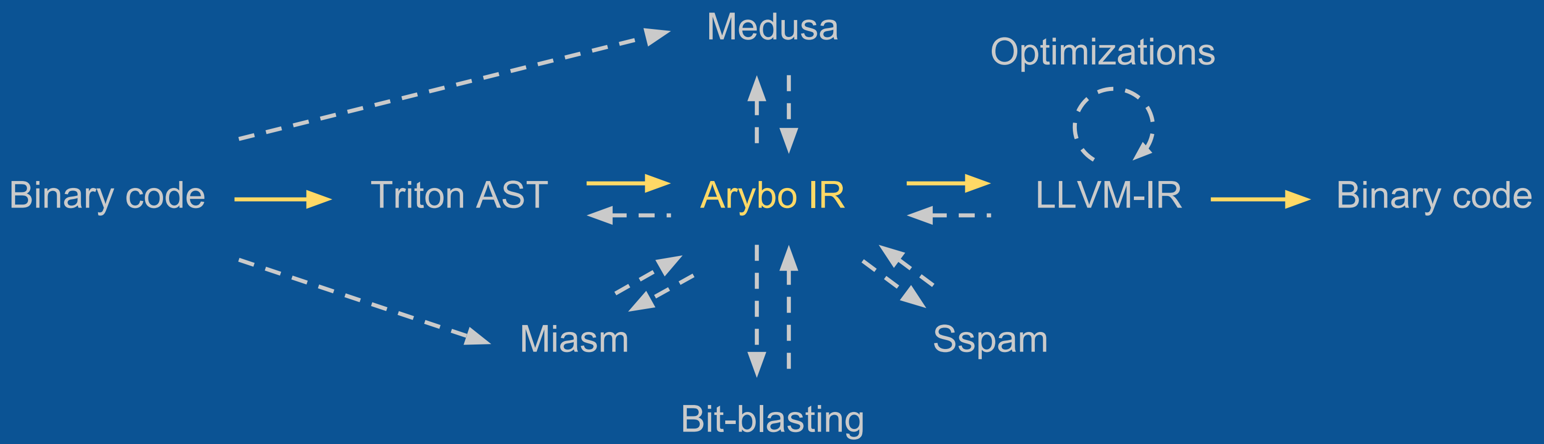
Step 4: Code Coverage - From a Paths Tree to a CFG?

- Two approaches
  - Custom algorithm *(not trivial)*
  - LLVM optimizations (-02) *(the lazy way)*



Step 5: Transformation to LLVM-IR

- In order to reconstruct a valid binary and apply paths merging
  - Move from our representation to the LLVM-IR
  - Arybo as crossroad



https://github.com/quarkslab/arybo

Step 6: Recompilation

- Based on the LLVM-IR we are able to:
    - Recompile a valid (and deobfuscated) code
    - Move to another architecture
    - Apply LLVM's analysis and optimizations

The Tigress Challenges

The Tigress Challenges

- Tigress
  - C Diversifier/Obfuscator
  - http://tigress.cs.arizona.edu
- Challenges
  - 35 VMs
  - $f(x) \rightarrow x'$
    - Function f is virtualized and we have to find the transformation algorithm

The Tigress Challenges

Challenge	Description	Number of binaries	Difficulty (1-10)	Script	Prize	Status
0000	One level of virtualization, random dispatch.	5	1	script	Certificate issued by DAPA	Solved
0001	One level of virtualization, superoperators, split instruction handlers.	5	2	script	Signed copy of Surreptitious Software.	Open
0002	One level of virtualization, bogus functions, implicit flow.	5	3	script	Signed copy of Surreptitious Software.	Open
0003	One level of virtualization, instruction handlers obfuscated with arithmetic encoding, virtualized function is split and the split parts merged.	5	2	script	Signed copy of Surreptitious Software.	Open
0004	Two levels of virtualization, implicit flow.	5	4	script	USD 100.00	Open
0005	One level of virtualization, one level of jitting, implicit flow.	5	4	script	USD 100.00	Open
0006	Two levels of jitting, implicit flow.	5	4	script	USD 100.00	Open

The Tigress Challenges

	Challenge-0	Challenge-1	Challenge-2	Challenge-3	Challenge-4
VM 0	3.85 seconds	9.20 seconds	3.27 seconds	4.26 seconds	1.58 seconds
VM 1	1.26 seconds	1.42 seconds	3.27 seconds	2.49 seconds	1.74 seconds
VM 2	6.58 seconds	2.02 seconds	2.63 seconds	4.85 seconds	3.82 seconds
VM 3	45.59 seconds	11.30 seconds	8.84 seconds	4.84 seconds	21.64 seconds
VM 4	361 seconds	315 seconds	588 seconds	8040 seconds	1680 seconds
	Few seconds to extract the equation and less than 200 MB of RAM used				
	Few minutes to extract the equation and ~4 GB of RAM used				
	Few minutes to extract the equation and ~5 GB of RAM used				
	Few minutes to extract the equation and ~9 GB of RAM used				
	Few minutes to extract the equation and ~21 GB of RAM used				
	Few hours to extract the equation and ~170 GB of RAM used				

Limitations

Limitations

- Our limitations are those of the symbolic execution
  - Code coverage of the virtualized function
    - Complexity of expressions
  - Multi-threading, IPC, asynchronous codes...

- Currently, we also have these limitations:
  - Loops reconstruction
  - Arrays reconstruction
    - Due to our concretization policy
  - Calls graph reconstruction

What Next?

What Next?

- Be able to determine on what designs of VM this approach works and doesn't
- Tests onto others protections

What Next?

- Be able to determine on what designs of VM this approach works and doesn't
- Tests onto others protections
  - Teasing: It’s working well on VMProtect

Demo

Conclusion

Conclusion

- Dynamic Taint Analysis + DSE
  - Powerful against VM based protections simplification
    - Automatic, independent from custom opcode, vpc, dispatcher, etc
- LLVM optimizations
  - Powerful for paths merging (and code simplification)
- Worked well for the Tigress protection
  - They (Tigress team) released a new protection
    - Code obfuscation against symbolic execution attacks ACSAC '16

Recommendation: Protections should also be applied onto the custom ISA instead of the process of the VM execution

Thanks - Questions?

https://triton.quarkslab.com
https://github.com/JonathanSalwan/Tigress_protection

Acknowledgements

- Adrien Guinet
  - Arybo support
- Romain Thomas
  - Ideas around path merging
- Gabriel Campana, Fred Raynal, Marion Videau
  - Review, proofreading