The Next Generation of Virtualization-based Obfuscators

Tim Blazytko

🐦 @mr_phrazer
⌂ synthesis.to

Moritz Schloegel

🐦 @m_u00d8
⌂ mschloegel.me

About Us

- Tim Blazytko
    - Chief Scientist, co-founder of emproof
    - designs software protections for embedded devices
    - trainer for (de)obfuscation and reverse engineering techniques

- Moritz Schloegel
    - last-year PhD student at CISPA Helmholtz Center
    - working with bugs by day (mostly fuzzing)
    - code deobfuscation by night

Setting the Scene

- VM-based obfuscation
- Attacks on VMs
- **Next-Gen**

Motivation

~~Prevent~~ **Complicate** reverse engineering attempts.

- intellectual property
- malicious payloads
- Digital Rights Management

Virtualization-based Obfuscation

Virtual Machines

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
  mov  edx, eax
  add  edx, ebx
  mov  eax, ebx
  mov  ebx, edx
  loop __secret_ip

mov eax, ebx
ret
```

Virtual Machines

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
  mov  edx, eax
  add  edx, ebx
  mov  eax, ebx
  mov  ebx, edx
  loop __secret_ip

mov eax, ebx
ret
```

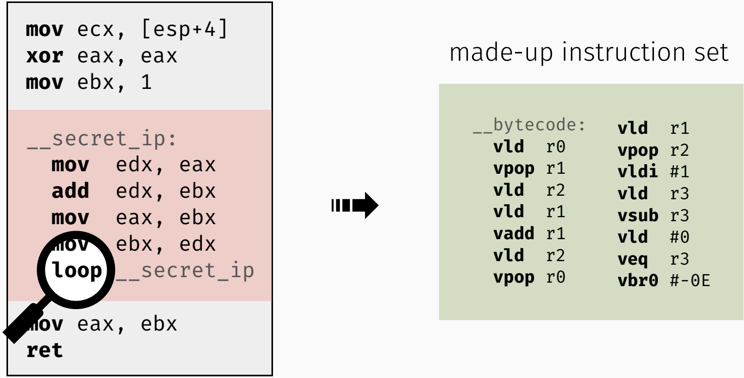
Virtual Machines

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
  mov  edx, eax
  add  edx, ebx
  mov  eax, ebx
  mov  ebx, edx
  loop __secret_ip

mov eax, ebx
ret
```

Virtual Machines



```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
   mov   edx, eax
   add   edx, ebx
   mov   eax, ebx
   mov   ebx, edx
   loop  __secret_ip

mov eax, ebx
ret
```

made-up instruction set

```
__bytecode:
   vld  r0
   vpop r1
   vld  r2
   vld  r1
   vadd r1
   vld  r2
   vpop r0
   vld  r1
   vpop r2
   vldi #1
   vld  r3
   vsub r3
   vld  #0
   veq  r3
   vbr0 #-0E
```

Virtual Machines

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
  push __bytecode
  call vm_entry

mov eax, ebx
ret
```

made-up instruction set

```
__bytecode:
  db 54 68 69 73 20 64 6f
  db 65 73 6e 27 74 20 6c
  db 6f 6f 6b 20 6c 69 6b
  db 65 20 61 6e 79 74 68
  db 69 6e 67 20 74 6f 20
  db 6d 65 2e de ad be ef
```

Virtual Machines

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
  push __bytecode
  call vm_entry

mov eax, ebx
ret
```

made-up instruction set

```
__bytecode:
  db 54 68 69 73 20 64 6f
  db 65 73 6e 27 74 20 6c
  db 6f 6f 6b 20 6c 69 6b
  db 65 20 61 6e 79 74 68
     69 6e 67 20 74 6f 20
      d 65 2e de ad be ef
```

Virtual Machines

Core Components

VM Entry/Exit	Context Switch: native context ⇔ virtual context
VM Dispatcher	Fetch–Decode–Execute loop
Handler Table	Individual VM ISA instruction semantics

Entry	Copy native context (registers, flags) to VM context.
Exit	Copy VM context back to native context.

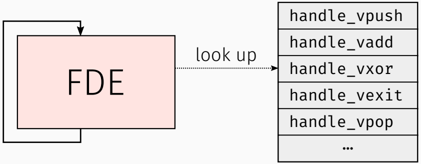
- Mapping from native to virtual registers is often 1:1.

Virtual Machines

Core Components

VM Entry/Exit	Context Switch: native context ⇔ virtual context
VM Dispatcher	Fetch–Decode–Execute loop
Handler Table	Individual VM ISA instruction semantics

1. Fetch and decode instruction
2. Forward virtual instruction pointer
3. Look up handler for opcode in handler table
4. Invoke handler

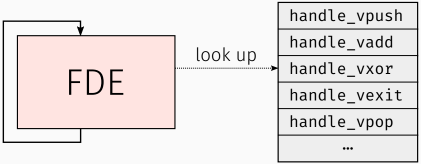


Virtual Machines

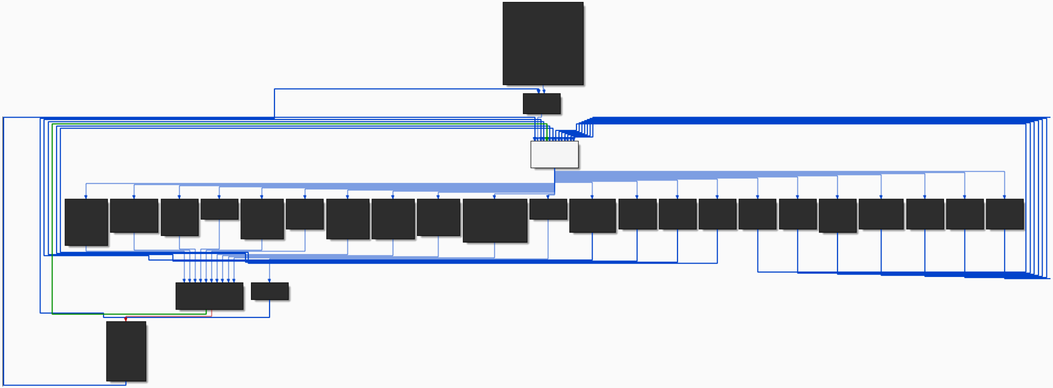
Core Components

VM Entry/Exit	Context Switch: native context ⇔ virtual context
VM Dispatcher	Fetch–Decode–Execute loop
Handler Table	Individual VM ISA instruction semantics

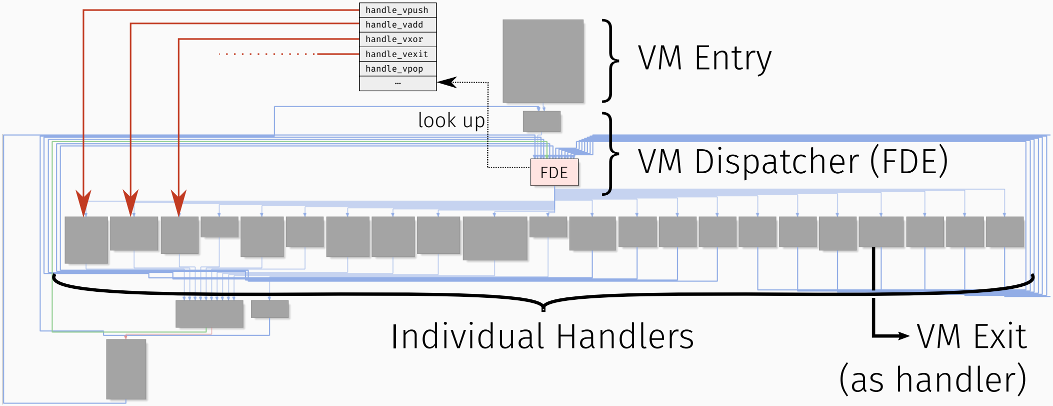
- Table of function pointers indexed by opcode
- One handler per virtual instruction
- Each handler decodes operands and updates VM context



Virtual Machines



Virtual Machines



Virtual Machines

```
__vm_dispatcher:
  mov   bl, [rsi]
  inc   rsi
  movzx rax, bl
  jmp   __handler_table[rax * 8]
```

VM Dispatcher

`rsi` – virtual instruction pointer
`rbp` – VM context

Virtual Machines

```
__vm_dispatcher:
  mov   bl, [rsi]
  inc   rsi
  movzx rax, bl
  jmp   __handler_table[rax * 8]
```

VM Dispatcher

`rsi` – virtual instruction pointer
`rbp` – VM context

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

Breaking Virtual Machine Obfuscation I

How to reconstruct the original code?

Breaking Virtual Machine Obfuscation I

How to reconstruct the original code?

1. understand VM architecture/context
2. reverse engineer handler semantics
3. write a disassembler for the bytecode
4. reconstruct VM control flow
5. reconstruct high-level code

Writing a VM Disassembler

opcode	register	register

Writing a VM Disassembler

opcode	register	register

```
0a 01 02 0b 01 05
```

Writing a VM Disassembler

opcode	register	register

```
0a 01 02 0b 01 05
```

```
add
```

Writing a VM Disassembler

opcode	register	register

```
0a 01 02 0b 01 05
```

```
add r1
```

Writing a VM Disassembler

opcode	register	register

```
0a 01 02 0b 01 05
```

```
add r1, r2
```

Writing a VM Disassembler

opcode	register	register

```
0a 01 02 0b 01 05
```

```
add r1, r2
mul
```

Writing a VM Disassembler

opcode	register	register

```
0a 01 02 0b 01 05
```

```
add r1, r2
mul r1
```

Writing a VM Disassembler

opcode	register	register

```
0a 01 02 0b 01 05
```

```
add r1, r2
mul r1, r5
```

Writing a VM Disassembler

opcode	register	register

```
0a 01 02 0b 01 05
```

```
add r1, r2
mul r1, r5
```

Writing a VM Disassembler

opcode	register	register

```
0a 01 02 0b 01 05
```

```
add r1, r2
mul r1, r5
```

VM computes **`(r1 + r2) * r5`**.

Virtual Machine Hardening

Virtual Machines

Hardening Technique #1 – Obfuscating individual VM components.

- Handlers are *conceptually simple*.

Virtual Machines

Hardening Technique #1 – Obfuscating individual VM components.

- Handlers are *conceptually simple*.
- Apply traditional code obfuscation transformations:
    - Substitution (`mov rax, rbx` ➟ `push rbx; pop rax`)
    - Opaque Predicates
    - Junk Code
    - …

```
mov eax, dword [rbp]
mov ecx, dword [rbp+4]
cmp r11w, r13w
sub rbp, 4
not eax
clc
cmc
cmp rdx, 0x28b105fa
not ecx
cmp r12b, r9b
```

Virtual Machines

Hardening Technique #2 – Duplicating VM handlers.

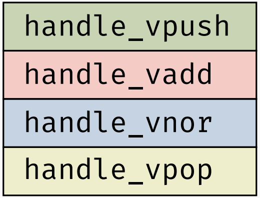
- Handler table is typically indexed using one byte (= 256 entries).

Virtual Machines

Hardening Technique #2 – Duplicating VM handlers.

- Handler table is typically indexed using one byte (= 256 entries).
- **Idea:** *Duplicate* existing handlers to populate full table.
- Use traditional obfuscation techniques to impede *code similarity* analyses.

Goal: Increase workload of reverse engineer.

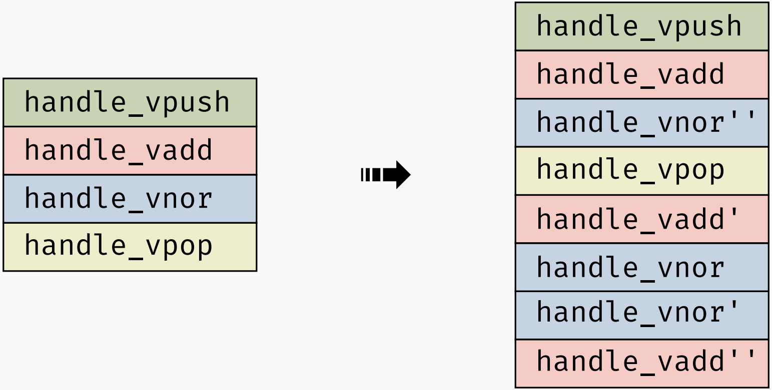


`handle_vpush`

`handle_vadd`

`handle_vnor`

`handle_vpop`



`handle_vpush`
`handle_vadd`
`handle_vnor`
`handle_vpop`

`handle_vpush`
`handle_vadd`
`handle_vnor''`
`handle_vpop`
`handle_vadd'`
`handle_vnor`
`handle_vnor'`
`handle_vadd''`

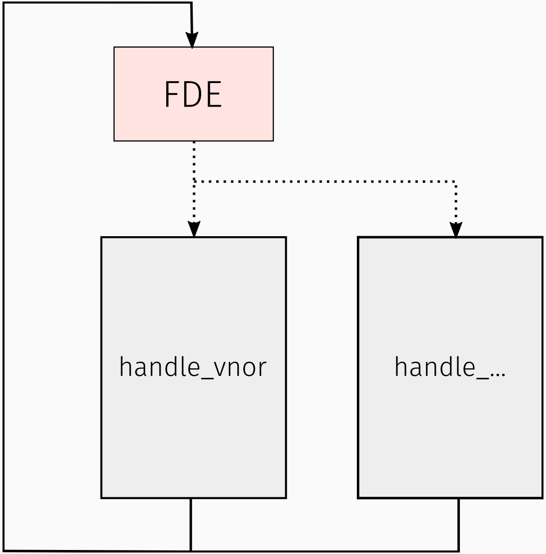
Virtual Machines

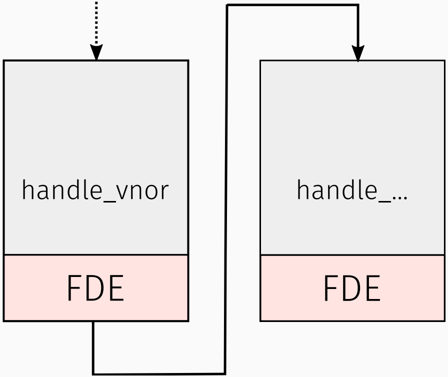
Hardening Technique #3 – No central VM dispatcher.

- A *central* VM dispatcher allows attacker to easily observe VM execution.
- **Idea:** Instead of branching to the central dispatcher, *inline* it into each handler.

Goal: No “single point of failure”.

(Themida, VMProtect Demo)


FDE
handle_vnor
handle_...


handle_vnor
FDE
handle_...
FDE

Programming Techniques
R. Morris Editor

Threaded Code

James R. Bell
Digital Equipment Corporation

The concept of "threaded code" is presented as an alternative to machine language code. Hardware and software realizations of it are given. In software it is realized as interpretive code not needing an interpreter. Extensions and optimizations are mentioned.

Key Words and Phrases: interpreter, machine code, time tradeoff, space tradeoff, compiled code, subroutine calls, threaded code

CR Categories: 4.12, 4.13, 6.33

Fig. 2 Flow of control: interpretive code.

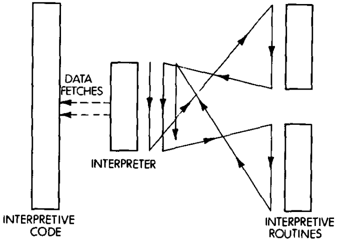
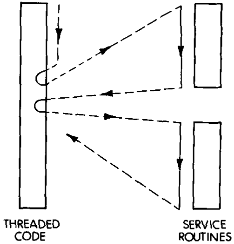


Fig. 3. Flow of control: threaded code.



Virtual Machines

Hardening Technique #4 – No explicit handler table.

- An *explicit* handler table easily reveals all VM handlers.

Virtual Machines

Hardening Technique #4 – No explicit handler table.

- An *explicit* handler table easily reveals all VM handlers.
- **Idea:** Instead of querying an explicit handler table,
  encode the next handler address in the VM instruction itself.

Goal: Hide location of handlers that have not been executed yet.

(VMProtect Full, SolidShield)

Virtual Machines

Hardening Technique #4 – No explicit handler table.

- An *explicit* handler table easily reveals all VM handlers.
- **Idea** table,
  e VM instruction itself.

opcode	op 0	op 1

Goal: Hide location of handlers that have not been executed yet.

(VMProtect Full, SolidShield)

Virtual Machines

Hardening Technique #4 – No explicit handler table.

- An *explicit* handler table easily reveals all VM handlers.
- **Idea**

opcode	op 0	op 1	next handler addr

Goal: Hide location of handlers that have not been executed yet.

(VMProtect Full, SolidShield)

SOFTWARE–PRACTICE AND EXPERIENCE, VOL. 11, 963–973 (1981)

Interpretation Techniques*

PAUL KLINT

Mathematical Centre, P.O. Box 4079, 1009AB Amsterdam, The Netherlands

SUMMARY

The relative merits of implementing high level programming languages by means of interpretation or compilation are discussed. The properties and the applicability of interpretation techniques known as classical interpretation, direct threaded code and indirect threaded code are described and compared.

KEY WORDS Interpretation versus compilation Interpretation techniques Instruction encoding Code generation Direct threaded code Indirect threaded code.

Virtual Machines

Hardening Technique #5 – Blinding VM bytecode.

- *Global analyses* on the bytecode possible, easy to patch instructions.

Virtual Machines

Hardening Technique #5 – Blinding VM bytecode.

- *Global analyses* on the bytecode possible, easy to patch instructions.
- **Idea:**
  - *Flow-sensitive* instruction decoding (“decryption” based on key register).
  - Custom decryption routine per handler, diversification.
  - Patching requires re-encryption of subsequent bytecode.

Goal: Hinder global analyses of bytecode and patching.

```
$operand \leftarrow [\mathbf{vIP} + 0]$

$context \leftarrow \mathrm{semantics}(context, operand)$
$next\_handler \leftarrow [\mathbf{vIP} + 4]$

$\mathbf{vIP} \leftarrow \mathbf{vIP} + 8$
$\mathbf{jmp}$ $next\_handler$
```

```
   $operand \leftarrow [\mathbf{vIP} + 0]$

🔑 $operand \leftarrow \text{unmangle}(operand, \mathbf{key})$
🔑 $\mathbf{key} \leftarrow \text{unmangle}'(\mathbf{key}, operand)$

   $context \leftarrow \text{semantics}(context, operand)$
   $next\_handler \leftarrow [\mathbf{vIP} + 4]$

🔑 $next\_handler \leftarrow \text{unmangle}''(next\_handler, \mathbf{key})$
🔑 $\mathbf{key} \leftarrow \text{unmangle}'''(\mathbf{key}, next\_handler)$

   $\mathbf{vIP} \leftarrow \mathbf{vIP} + 8$
   $\mathbf{jmp}\ next\_handler$
```

Breaking Virtual Machine Obfuscation II

How to deal with hardened VMs?

- locate **VM entry** and **bytecode**
- **simplify handlers** with program analyses techniques
- write a **control-flow sensitive disassembler**[1] and reconstruct high-level code

---

[1] `https://synthesis.to/2021/10/21/vm_based_obfuscation.html`

Automated Attacks on VMs

Instruction Removal

Compiler Optimizations

```
mov eax, 0xdead
mov eax, 0x1234
not eax
push eax
mov eax, 0x5678
mov ecx, ecx
add eax, 0x1111
add ecx, 0x0
mov edx, eax
pop eax
not eax
ret
```

Compiler Optimizations

```
mov eax, 0xdead
mov eax, 0x1234
not eax
push eax
mov eax, 0x5678
mov ecx, ecx
add eax, 0x1111
add ecx, 0x0
mov edx, eax
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
mov eax, 0x5678
×
add eax, 0x1111
×
mov edx, eax
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
mov eax, 0x5678
add eax, 0x1111
×
mov edx, eax
pop eax
not eax
ret
```

Dead Code Elimination

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
mov eax, 0x5678
×
add eax, 0x1111
×
mov edx, eax
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
mov eax, 0x5678
×
add eax, 0x1111
×
mov edx, eax
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
×
×
mov eax, 0x6789
×
mov edx, eax
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
×
×
mov edx, eax
pop eax
not eax
ret
```

Constant Folding

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
×
×
mov eax, 0x6789
×
mov edx, eax
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
×
×
mov eax, 0x6789
×
mov edx, eax
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
×
×
×
×
mov edx, 0x6789
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
×
```

Constant Propagation

```
×
mov edx, 0x6789
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
×
×
×
×
mov edx, 0x6789
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
push eax
×
×
×
×
mov edx, 0x6789
pop eax
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
×
×
×
×
×
mov edx, 0x6789
×
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
not eax
×
×
×
×
×
mov edx, 0x6789
×
not eax
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
×
×
×
×
×
×
mov edx, 0x6789
×
×
ret
```

Compiler Optimizations

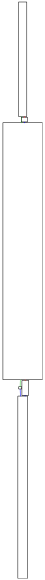
```
×
mov eax, 0x1234
×
×
×
```

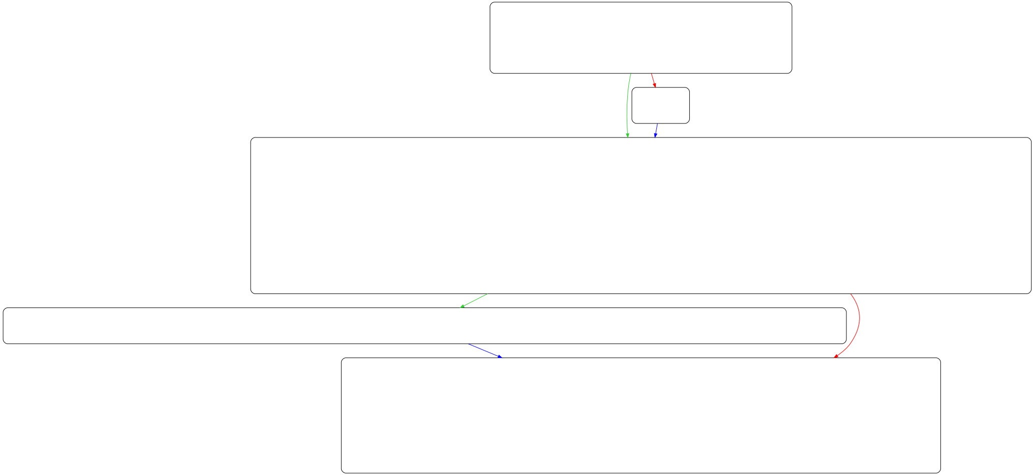
Peephole Optimization

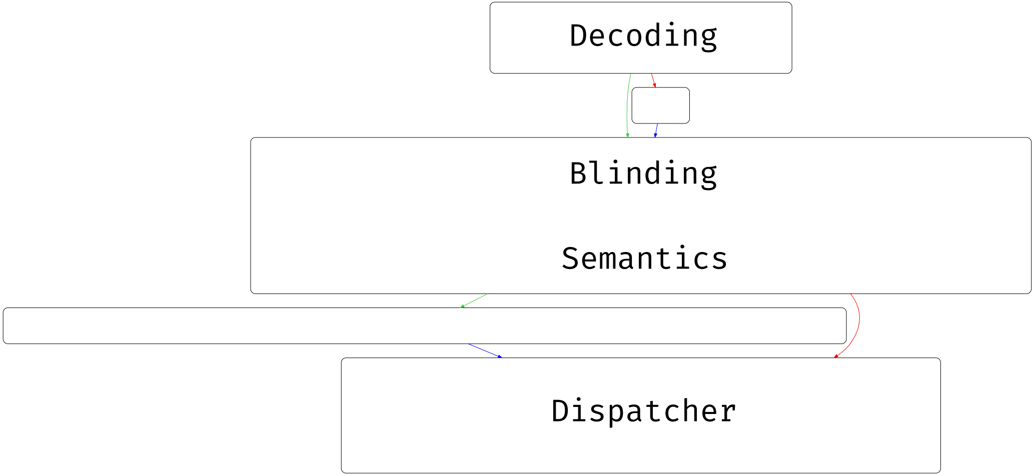
```
×
mov edx, 0x6789
×
×
ret
```

Compiler Optimizations

```
×
mov eax, 0x1234
×
×
×
×
×
×
mov edx, 0x6789
×
×
ret
```





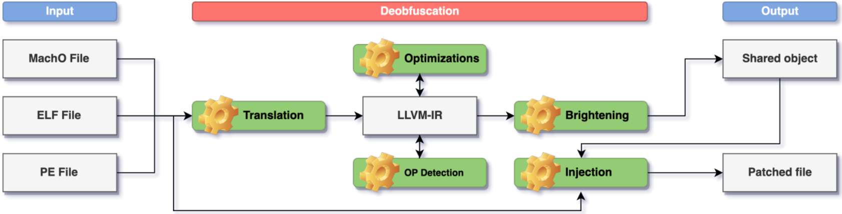

Decoding
Blinding
Semantics
Dispatcher

SATURN

Software Deobfuscation Framework Based on LLVM

Peter Garba*
Thales, DIS - Cybersecurity
Munich, Germany
peter.garba@thalesgroup.com

Matteo Favaro
Zimperium, Mobile Security
Noale, Italy
matteo.favaro@reversing.software



Symbolic Execution

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
• mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor**
(with flag side-effects)

rcx ← [rbp]

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
• mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

$$\begin{aligned} rcx &\leftarrow [rbp] \\ rbx &\leftarrow [rbp + 4] \end{aligned}$$

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
• not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor**
(with flag side-effects)

$$\begin{aligned} \mathrm{rcx} &\leftarrow [\mathrm{rbp}] \\ \mathrm{rbx} &\leftarrow [\mathrm{rbp}+4] \\ \mathrm{rcx} &\leftarrow \neg\,\mathrm{rcx} = \neg\,[\mathrm{rbp}] \end{aligned}$$

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
• not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor**
(with flag side-effects)

$$\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \neg\,\text{rcx} = \neg\,[\text{rbp}] \\
\text{rbx} &\leftarrow \neg\,\text{rbx} = \neg\,[\text{rbp} + 4]
\end{aligned}$$

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
• and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

$$\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \neg\, \text{rcx} = \neg\, [\text{rbp}] \\
\text{rbx} &\leftarrow \neg\, \text{rbx} = \neg\, [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \text{rcx} \land \text{rbx} \\
&\qquad = (\neg\, [\text{rbp}]) \land (\neg\, [\text{rbp} + 4])
\end{aligned}$$

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
• and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor**
(with flag side-effects)

$$\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \neg\,\text{rcx} = \neg\,[\text{rbp}] \\
\text{rbx} &\leftarrow \neg\,\text{rbx} = \neg\,[\text{rbp} + 4] \\
\text{rcx} &\leftarrow \text{rcx} \wedge \text{rbx} \\
&\qquad = (\neg\,[\text{rbp}]) \wedge (\neg\,[\text{rbp} + 4]) \\
&\qquad = [\text{rbp}] \downarrow [\text{rbp} + 4]
\end{aligned}$$

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
• mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

$$
\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp}+4] \\
\text{rcx} &\leftarrow \neg\,\text{rcx} = \neg\,[\text{rbp}] \\
\text{rbx} &\leftarrow \neg\,\text{rbx} = \neg\,[\text{rbp}+4] \\
\text{rcx} &\leftarrow \text{rcx} \wedge \text{rbx} \\
&\qquad = (\neg\,[\text{rbp}]) \wedge (\neg\,[\text{rbp}+4]) \\
&\qquad = [\text{rbp}] \downarrow [\text{rbp}+4] \\
[\text{rbp}+4] &\leftarrow \text{rcx} = [\text{rbp}] \downarrow [\text{rbp}+4]
\end{aligned}
$$

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
• pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

$$
\begin{aligned}
\mathsf{rcx} &\leftarrow [\mathsf{rbp}] \\
\mathsf{rbx} &\leftarrow [\mathsf{rbp}+4] \\
\mathsf{rcx} &\leftarrow \neg\,\mathsf{rcx} = \neg\,[\mathsf{rbp}] \\
\mathsf{rbx} &\leftarrow \neg\,\mathsf{rbx} = \neg\,[\mathsf{rbp}+4] \\
\mathsf{rcx} &\leftarrow \mathsf{rcx} \wedge \mathsf{rbx} \\
&\qquad = (\neg\,[\mathsf{rbp}]) \wedge (\neg\,[\mathsf{rbp}+4]) \\
&\qquad = [\mathsf{rbp}] \downarrow [\mathsf{rbp}+4] \\
[\mathsf{rbp}+4] &\leftarrow \mathsf{rcx} = [\mathsf{rbp}] \downarrow [\mathsf{rbp}+4] \\
\\
\mathsf{rsp} &\leftarrow \mathsf{rsp} - 4 \\
[\mathsf{rsp}] &\leftarrow \mathsf{flags}
\end{aligned}
$$

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
• pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

$$
\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \neg\,\text{rcx} = \neg\,[\text{rbp}] \\
\text{rbx} &\leftarrow \neg\,\text{rbx} = \neg\,[\text{rbp} + 4] \\
\text{rcx} &\leftarrow \text{rcx} \wedge \text{rbx} \\
&\qquad = (\neg\,[\text{rbp}]) \wedge (\neg\,[\text{rbp} + 4]) \\
&\qquad = [\text{rbp}] \downarrow [\text{rbp} + 4] \\
[\text{rbp} + 4] &\leftarrow \text{rcx} = [\text{rbp}] \downarrow [\text{rbp} + 4]
\end{aligned}
$$

$$
\begin{aligned}
\text{rsp} &\leftarrow \text{rsp} - 4 \\
[\text{rsp}] &\leftarrow \text{flags} \\
[\text{rbp}] &\leftarrow [\text{rsp}] = \text{flags} \\
\text{rsp} &\leftarrow \text{rsp} + 4
\end{aligned}
$$

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
• jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

$$\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \neg\,\text{rcx} = \neg\,[\text{rbp}] \\
\text{rbx} &\leftarrow \neg\,\text{rbx} = \neg\,[\text{rbp} + 4] \\
\text{rcx} &\leftarrow \text{rcx} \wedge \text{rbx} \\
&\qquad = (\neg\,[\text{rbp}]) \wedge (\neg\,[\text{rbp} + 4]) \\
&\qquad = [\text{rbp}] \downarrow [\text{rbp} + 4] \\
[\text{rbp} + 4] &\leftarrow \text{rcx} = [\text{rbp}] \downarrow [\text{rbp} + 4]
\end{aligned}$$

$$\begin{aligned}
\text{rsp} &\leftarrow \text{rsp} - 4 \\
[\text{rsp}] &\leftarrow \text{flags} \\
[\text{rbp}] &\leftarrow [\text{rsp}] = \text{flags} \\
\text{rsp} &\leftarrow \text{rsp} + 4
\end{aligned}$$

Symbolic Execution: A Syntactic Approach

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

$$
\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \neg\,\text{rcx} = \neg\,[\text{rbp}] \\
\text{rbx} &\leftarrow \neg\,\text{rbx} = \neg\,[\text{rbp} + 4] \\
\text{rcx} &\leftarrow \text{rcx} \wedge \text{rbx} \\
&= [\text{rbp}] \downarrow [\text{rbp} + 4] \\
[\text{rbp} + 4] &\leftarrow \text{rcx} = [\text{rbp}] \downarrow [\text{rbp} + 4]
\end{aligned}
$$

$$
\begin{aligned}
\text{rsp} &\leftarrow \text{rsp} - 4 \\
[\text{rsp}] &\leftarrow \text{flags} \\
[\text{rbp}] &\leftarrow [\text{rsp}] = \text{flags} \\
\text{rsp} &\leftarrow \text{rsp} + 4
\end{aligned}
$$

$$[\text{rbp} + 4] \leftarrow ([\text{rbp}] \downarrow [\text{rbp} + 4])$$

Program Synthesis

Program Synthesis: A Semantic Approach

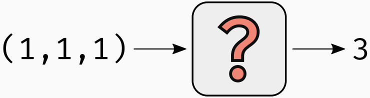
We use f as a black-box:

$$f(x, y, z) := (((x \oplus y) + ((x \wedge y) \cdot 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \cdot 2)) \wedge z)$$

Program Synthesis: A Semantic Approach

We use f as a black-box:

$$f(x,y,z) := (((x \oplus y) + ((x \wedge y) \cdot 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \cdot 2)) \wedge z)$$

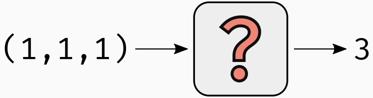


`(1,1,1)` → 3

Program Synthesis: A Semantic Approach

We use f as a black-box:

$$f(x,y,z) := (((x \oplus y) + ((x \wedge y) \cdot 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \cdot 2)) \wedge z)$$

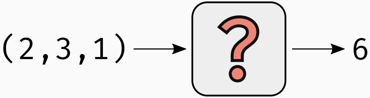


$(1,1,1) \rightarrow 3$

Program Synthesis: A Semantic Approach

We use f as a black-box:

$$f(x,y,z) := (((x \oplus y) + ((x \wedge y) \cdot 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \cdot 2)) \wedge z)$$

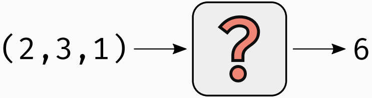


$(1,1,1) \rightarrow 3$

Program Synthesis: A Semantic Approach

We use f as a black-box:

$$f(x,y,z) := (((x \oplus y) + ((x \wedge y) \cdot 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \cdot 2)) \wedge z)$$



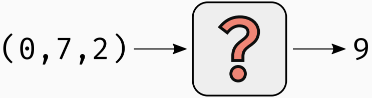
$(1,1,1) \rightarrow 3$

$(2,3,1) \rightarrow 6$

Program Synthesis: A Semantic Approach

We use f as a black-box:

$$f(x, y, z) := (((x \oplus y) + ((x \wedge y) \cdot 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \cdot 2)) \wedge z)$$



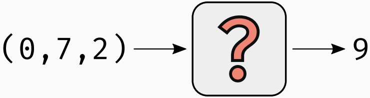
$(1, 1, 1) \rightarrow 3$

$(2, 3, 1) \rightarrow 6$

Program Synthesis: A Semantic Approach

We use f as a black-box:

$$f(x, y, z) := (((x \oplus y) + ((x \wedge y) \cdot 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \cdot 2)) \wedge z)$$



$(1, 1, 1) \rightarrow 3$

$(2, 3, 1) \rightarrow 6$

$(0, 7, 2) \rightarrow 9$

Program Synthesis: A Semantic Approach

We use f as a black-box:

$$f(x, y, z) := (((x \oplus y) + ((x \wedge y) \cdot 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \cdot 2)) \wedge z)$$

$$(1, 1, 1) \rightarrow 3$$

$$(2, 3, 1) \rightarrow 6$$

$$(0, 7, 2) \rightarrow 9$$

We **learn** a function h that has the same I/O behavior.

Program Synthesis: A Semantic Approach

We use f as a black-box:

$$f(x,y,z) := (((x \oplus y) + ((x \wedge y) \cdot 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \cdot 2)) \wedge z)$$

$$\boxed{h(x,y,z) := x + y + z}$$

$\to 3$

$(2,3,1) \to 6$

$(0,7,2) \to 9$

We **learn** a function h that has the same I/O behavior.

Synthesis Light: Code Book Attacks

Synthesis Light: Code Book Attacks

VM ISA

- $x + y$
- $x - y$
- $x \wedge y$
- $x \vee y$
- $x \oplus y$

- **predictable** set of handler semantics

Synthesis Light: Code Book Attacks

VM ISA

- $x + y$
- $x - y$
- $x \wedge y$
- $x \vee y$
- $x \oplus y$

Lookup Table

$(5, 3)$	→	8:	$x + y$
$(5, 3)$	→	2:	$x - y$
$(5, 3)$	→	1:	$x \wedge y$
$(5, 3)$	→	7:	$x \vee y$
$(5, 3)$	→	6:	$x \oplus y$

- **predictable** set of handler semantics
- **pre-computed lookup tables** of I/O samples

Synthesis Light: Code Book Attacks

VM ISA

- $x + y$
- $x - y$
- $x \wedge y$
- $x \vee y$
- $x \oplus y$

Lookup Table

$(5, 3)$	$\rightarrow$	8:	$x + y$
$(5, 3)$	$\rightarrow$	2:	$x - y$
$(5, 3)$	$\rightarrow$	1:	$x \wedge y$
$(5, 3)$	$\rightarrow$	7:	$x \vee y$
$(5, 3)$	$\rightarrow$	6:	$x \oplus y$

- **predictable** set of handler semantics
- **pre-computed lookup tables** of I/O samples
- SMT solvers to prove **semantic equivalence**

Attack Surface

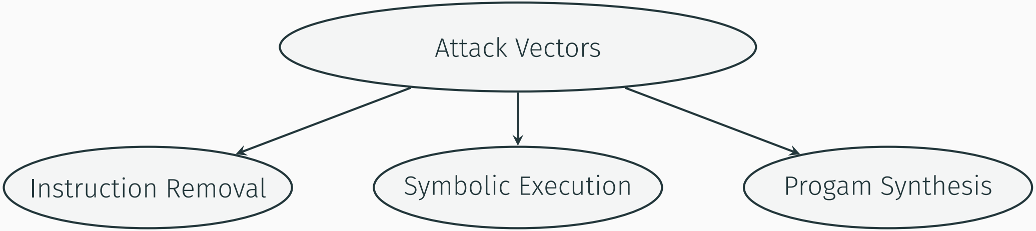
Shortcomings of VMs

- **predictable** instruction semantics with **meaningful** mnemonics
    - vulnerable to synthesis-based attacks
    - facilitates writing **disassemblers**

Shortcomings of VMs

- **predictable** instruction semantics with **meaningful** mnemonics
  - vulnerable to synthesis-based attacks
  - facilitates writing **disassemblers**
- VM components are **independent** of each other
  - isolated analysis possible
  - obfuscation limited to **local** constructs (e.g., handler level)

VM Attack Landscape



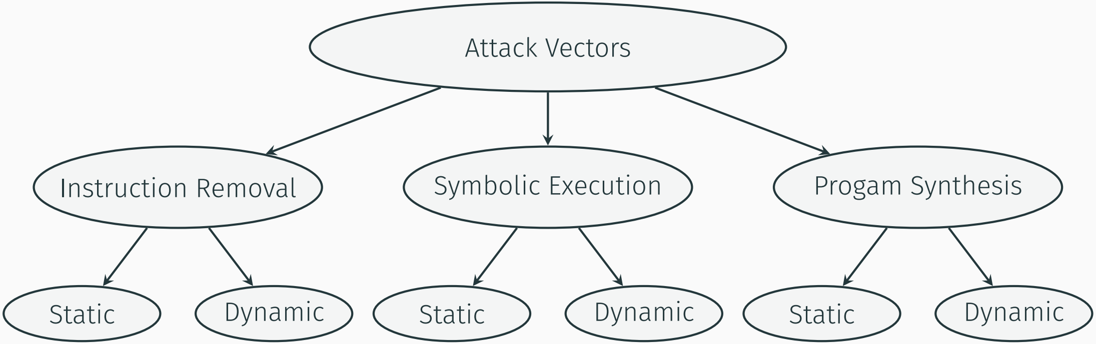
Attack Vectors

Instruction Removal

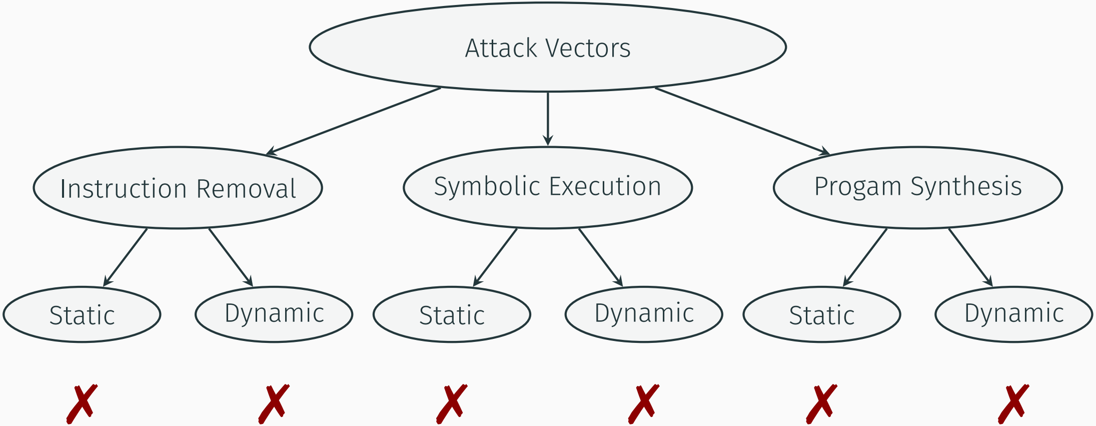
Symbolic Execution

Progam Synthesis

VM Attack Landscape



VM Attack Landscape



Attack Vectors

Instruction Removal

Symbolic Execution

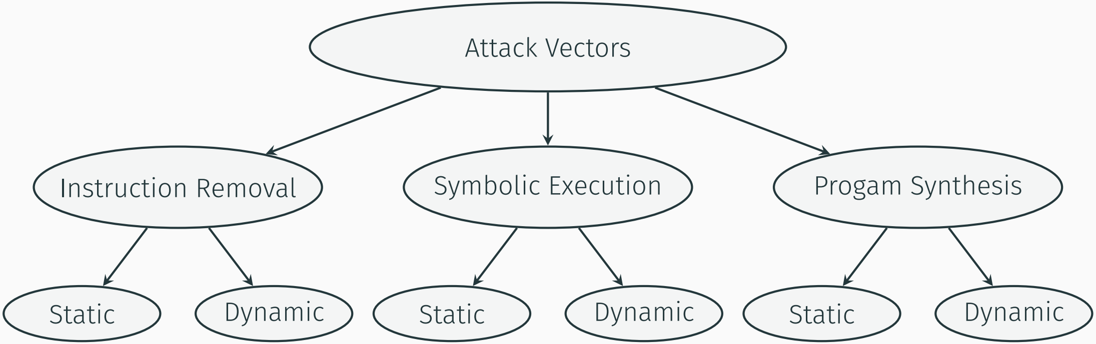
Progam Synthesis

Static Dynamic Static Dynamic Static Dynamic

✗ ✗ ✗ ✗ ✗ ✗

Next-Gen VM-based Obfuscators

Design Goals


Attack Vectors

Instruction Removal

Symbolic Execution

Progam Synthesis

Static

Dynamic

Static

Dynamic

Static

Dynamic

Design Principles

Design Principles

Design Principle #1 – Complex and target-specific instruction sets.

Design Principles

Design Principle #1 – Complex and target-specific instruction sets.

- handler semantics are based on **instruction sequences from the target program**

Design Principles

Design Principle #1 – Complex and target-specific instruction sets.

- handler semantics are based on **instruction sequences from the target program**
- **complex** handler **semantics**
  - introduce diversity
  - provide resilience against syntesis-based attacks

Design Principles

Design Principle #1 – Complex and target-specific instruction sets.

- handler semantics are based on **instruction sequences from the target program**
- **complex** handler **semantics**
    - introduce diversity
    - provide resilience against syntesis-based attacks
- can be **data-flow** dependent

Design Principles

Design Principle #1 – Complex and target-specific instruction sets.

- handler semantics are based on **instruction sequences from the target program**

No meaningful instruction mnemonics for VM disassemblers

  - introduce diversity
  - provide resilience against syntesis-based attacks
- can be **data-flow** dependent

Design Principles

Design Principle #2 – Intertwining VM components.

Design Principles

Design Principle #2 – Intertwining VM components.

- **interlocking** of handlers & semantics to enforce a **cross-handler** analysis
    - mixed Boolean-Arithmetic encodings across handlers
    - dataflow-dependent or multi-threaded opaque predicates
    - merged handler semantics

Design Principles

Design Principle #2 – Intertwining VM components.

- **interlocking** of handlers & semantics to enforce a **cross-handler** analysis
  - mixed Boolean-Arithmetic encodings across handlers
  - dataflow-dependent or multi-threaded opaque predicates
  - merged handler semantics
- analysis **effort rises** enormously

Design Principles

Design Principle #2 – Intertwining VM components.

- **interlocking** of handlers & semantics to enforce a **cross-handler** analysis
    - mi
    - dataflow-dependent or multi-threaded opaque predicates
    - merged handler semantics
- analysis **effort rises** enormously

Analysis tools reach their limits

Loki

Loki

- academic prototype of next-gen VM
- industry shifts towards novel VM designs
- paper at USENIX Sec'22: "Loki: Hardening Code Obfuscation Against Automated Attacks"
  `https://www.usenix.org/system/files/sec22-schloegel.pdf`

LOKI: Hardening Code Obfuscation Against Automated Attacks

Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann
Julius Basler, Thorsten Holz, Ali Abbasi

Ruhr-Universität Bochum, Germany

Current VM Handlers

opcode	register	register

```
0a 01 02        add r1, r2
0b 01 05        mul r1, r5
```

Current VM Handlers

opcode	register	register

`0a 01 02`	`add r1, r2`	$f(x, y) := x + y$
`0b 01 05`	`mul r1, r5`	$g(x, y) := x * y$

- handler can be represented as mathematical functions

Current VM Handlers

opcode	register	register

`0a 01 02`	`add r1, r2`	$f(x, y) := x + y$
`0b 01 05`	`mul r1, r5`	$g(x, y) := x * y$
`a2 03 ??`	`shl r3, 0xff`	

- handler can be represented as mathematical functions

Current VM Handlers

opcode	register	register

`0a 01 02`	`add r1, r2`	$f(x, y) := x + y$
`0b 01 05`	`mul r1, r5`	$g(x, y) := x * y$
`a2 03 ??`	`shl r3, 0xff`	

- handler can be represented as mathematical functions

Current VM Handlers

opcode	register	register	constant

`0a 01 02 00`	`add r1, r2`	$f(x, y) := x + y$
`0b 01 05 00`	`mul r1, r5`	$g(x, y) := x * y$
`a2 03 ?? ff`	`shl r3, 0xff`	

- handler can be represented as mathematical functions

Current VM Handlers

opcode	register	register	constant

`0a 01 02 00`	`add r1, r2`	$f(x, y, c) := x + y$
`0b 01 05 00`	`mul r1, r5`	$g(x, y, c) := x * y$
`a2 03 ?? ff`	`shl r3, 0xff`	

- handler can be represented as mathematical functions

Current VM Handlers

opcode	register	register	constant

`0a 01 02 00`	`add r1, r2`	$f(x, y, c) := x + y$
`0b 01 05 00`	`mul r1, r5`	$g(x, y, c) := x * y$
`a2 03 ?? ff`	`shl r3, 0xff`	$h(x, y, c) := x << c$

- handler can be represented as mathematical functions

Current VM Handlers

opcode	register	register	constant

`0a 01 02 00`	`add r1, r2`	$f(x,y,c) := x + y$
`0b 01 05 00`	`mul r1, r5`	$g(x,y,c) := x * y$
`a2 03 ?? ff`	`shl r3, 0xff`	$h(x,y,c) := x << c$

- handler can be represented as mathematical functions
- instruction semantics refer to the handler’s actual computation

Can we do better?

Merging Instruction Semantics

$f(x, y, c) := x + y$

$g(x, y, c) := x - y << c$

Merging Instruction Semantics

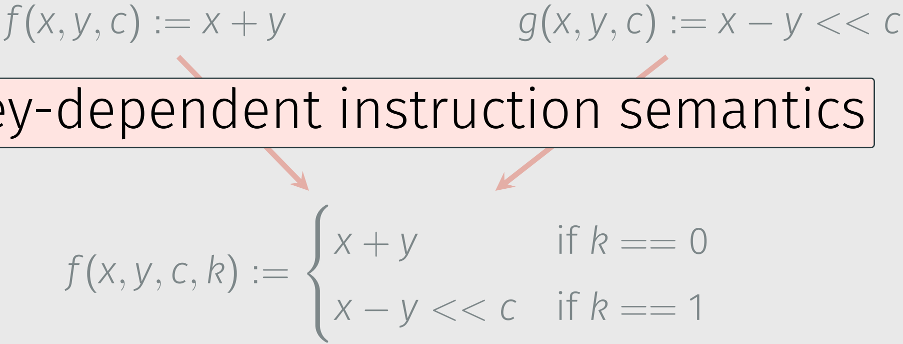
$$\begin{tikzcd}
f(x, y, c) := x + y \arrow[dr] & & g(x, y, c) := x - y << c \arrow[dl] \\
& f(x, y, c, k) := \begin{cases} x + y & \text{if } k == 0 \\ x - y << c & \text{if } k == 1 \end{cases} &
\end{tikzcd}$$

Merging Instruction Semantics

$$f(x, y, c) := x + y \qquad\qquad g(x, y, c) := x - y << c$$

$$f(x, y, c, k) := \begin{cases} x + y & \text{if } k == 0 \\ x - y << c & \text{if } k == 1 \end{cases}$$

Merging Instruction Semantics



$f(x, y, c) := x + y$

$g(x, y, c) := x - y << c$

Key-dependent instruction semantics

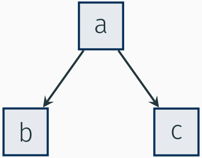
$$f(x, y, c, k) := \begin{cases} x + y & \text{if } k == 0 \\ x - y << c & \text{if } k == 1 \end{cases}$$

Polynomial Encodings and Branch-free Code

$$f(x, y, c, k) := \begin{cases} x + y & \text{if } k == 0 \\ x - y << c & \text{if } k == 1 \end{cases}$$

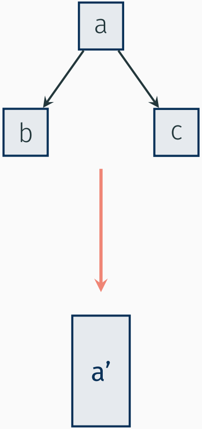
Polynomial Encodings and Branch-free Code

$$f(x, y, c, k) := \begin{cases} x + y & \text{if } k == 0 \\ x - y << c & \text{if } k == 1 \end{cases}$$



Polynomial Encodings and Branch-free Code

$$f(x, y, c, k) := \begin{cases} x + y & \text{if } k == 0 \\ x - y << c & \text{if } k == 1 \end{cases}$$

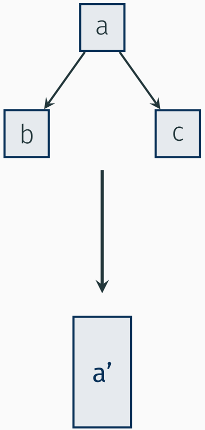


Polynomial Encodings and Branch-free Code

$$f(x, y, c, k) := \begin{cases} x + y & \text{if } k == 0 \\ x - y << c & \text{if } k == 1 \end{cases}$$

$$\Big\Updownarrow \ \textit{equal}$$

$$\begin{aligned} f(x, y, c, k) := \quad & (k == 0) \cdot x + y \\ + \quad & (k == 1) \cdot x - y << c \end{aligned}$$

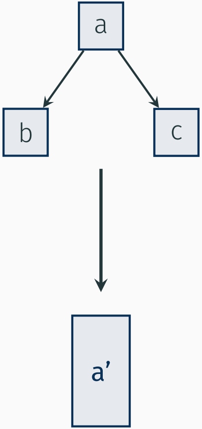


Polynomial Encodings and Branch-free Code

$$f(x,y,c,k) := \begin{cases} x+y & \text{if } k == 0 \\ x-y << c & \text{if } k == 1 \end{cases}$$

$$\Updownarrow$$

$$\begin{aligned} f(x,y,c,k) := &\quad (k == 0) \cdot x+y \\ + &\quad (k == 1) \cdot x-y << c \end{aligned}$$



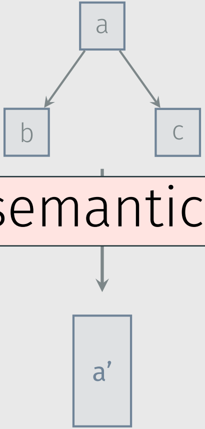
Polynomial Encodings and Branch-free Code

$$f(x, y, c, k) := \begin{cases} x + y & \text{if } k == 0 \\ x - y << c & \text{if } k == 1 \end{cases}$$

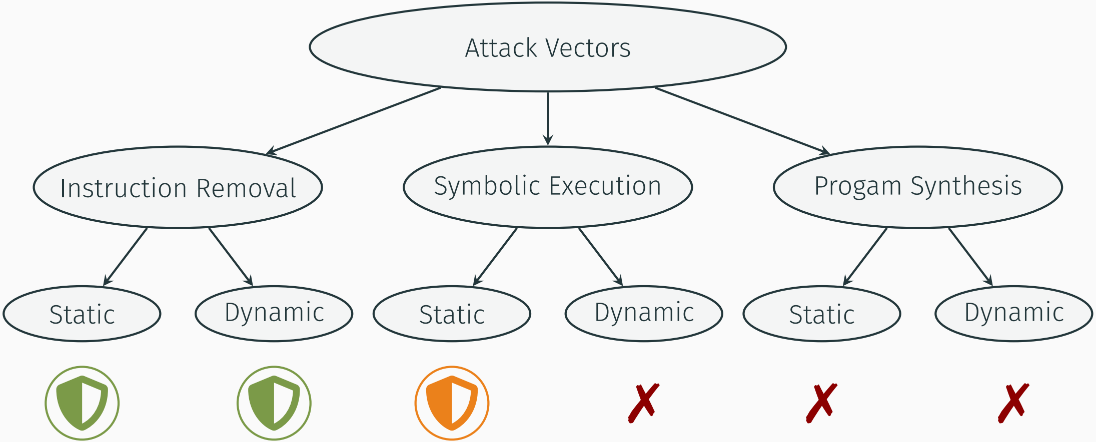
$$\Updownarrow$$

Interlocking of instruction semantics

$$f(x, y, c, k) := \quad (k == 0) \cdot x + y \quad + \quad (k == 1) \cdot x - y << c$$



Polynomial Encodings



Attack Vectors

Instruction Removal

Symbolic Execution

Progam Synthesis

Static

Dynamic

Static

Dynamic

Static

Dynamic

✗

✗

✗

Hardening Key Selection

$$
\begin{aligned}
f(x, y, c, k) := & \quad (k == 0) \cdot x + y \\
& + \quad (k == 1) \cdot x - y << c
\end{aligned}
$$

Hardening Key Selection

$$\begin{aligned} f(x, y, c, k) := \quad & (n \mod k == 0) \quad \cdot \quad x + y \\ + \quad & (k^2 == q \mod m) \quad \cdot \quad x - y << c \end{aligned}$$

Hardening Key Selection

Factorization

$$\begin{aligned} f(x, y, c, k) :=& \quad (n \mod k == 0) \quad \cdot \quad x + y \\ &+ \quad (k^2 == q \mod m) \quad \cdot \quad x - y << c \end{aligned}$$

Hardening Key Selection

Factorization

$$
\begin{aligned}
f(x, y, c, k) := \quad & \left(n \mod k == 0\right) \cdot x + y \\
+ \quad & \left(k^2 == q \mod m\right) \cdot x - y << c
\end{aligned}
$$

Quadratic Residues

Hardening Key Selection

Factorization

SMT-hard encodings for instruction selection

$$+ \quad (k^2 == q \mod m) \quad \cdot \quad x - y << c$$

Quadratic Residues

Point Functions

Partial point functions for key selection

$$\begin{aligned} f(x, y, c, k) := \quad & (n \mod k == 0) \quad \cdot \quad x + y \\ + \quad & (k^2 == q \mod m) \quad \cdot \quad x - y << c \end{aligned}$$

Point Functions

Partial point functions for key selection

$$\begin{aligned} f(x, y, c, k) := \quad & (n \mod k == 0) \cdot x + y \\ + \quad & \mathit{pf(k)} \cdot x - y << c \end{aligned}$$

Point Functions

Partial point functions for key selection

$$\begin{aligned} f(x, y, c, k) := & \quad (n \mod k == 0) \cdot x + y \\ & + \quad pf(k) \cdot x - y << c \end{aligned}$$

$$pf(k) := ((\mathrm{0xff} \wedge k) \oplus \mathrm{0xcd}) \cdot \mathrm{0x28cbfbeb9a020a33}$$

Point Functions

Partial point functions for key selection

$$\begin{aligned} f(x, y, c, k) := & \quad (n \mod k == 0) & \cdot \quad & x + y \\ + & \quad \boldsymbol{pf(k)} & \cdot \quad & x - y << c \end{aligned}$$

$$\boldsymbol{pf(k)} := ((\mathrm{0xff} \wedge k) \oplus \mathrm{0xcd}) \cdot \mathrm{0x28cbfbeb9a020a33}$$

$pf(\mathrm{0x1336}) = 1$

Point Functions

Partial point functions for key selection

$$\begin{aligned} f(x, y, c, k) := & \quad (n \mod k == 0) \cdot x + y \\ + & \quad \mathit{pf(k)} \cdot x - y << c \end{aligned}$$

$$\mathit{pf(k)} := ((0\text{xff} \wedge k) \oplus 0\text{xcd}) \cdot 0\text{x28cbfbeb9a020a33}$$

$$pf(0\text{x1336}) = 1$$

✓

Point Functions

Partial point functions for key selection

$$\begin{aligned} f(x, y, c, k) := \quad & (n \mod k == 0) & \cdot \quad & x + y \\ + \quad & \boldsymbol{pf(k)} & \cdot \quad & x - y << c \end{aligned}$$

$$\boldsymbol{pf(k)} := ((\mathrm{0xff} \wedge k) \oplus \mathrm{0xcd}) \cdot \mathrm{0x28cbfbeb9a020a33}$$

$pf(\mathrm{0x1336}) = 1$ $\quad$ $pf(\mathrm{0xabcd}) = 0$

✓

Point Functions

Partial point functions for key selection

$$\begin{aligned} f(x,y,c,k) :=\ & (n \mod k == 0) \cdot x+y \\ +\ & pf(k) \cdot x-y << c \end{aligned}$$

$$\boldsymbol{pf(k)} := ((\mathrm{0xff} \land k) \oplus \mathrm{0xcd}) \cdot \mathrm{0x28cbfbeb9a020a33}$$

$pf(\mathrm{0x1336}) = 1$ $pf(\mathrm{0xabcd}) = 0$

✓ ✗

Point Functions

Partial point functions for key selection

$$\begin{aligned} f(x,y,c,k) := \quad & (n \mod k == 0) && \cdot \quad x + y \\ + \quad & \mathit{pf}(k) && \cdot \quad x - y << c \end{aligned}$$

$$\boldsymbol{\mathit{pf}(k)} := ((\mathrm{0xff} \wedge k) \oplus \mathrm{0xcd}) \cdot \mathrm{0x28cbfbeb9a020a33}$$

$\mathit{pf}(\mathrm{0x1336}) = 1$ $\quad$ $\mathit{pf}(\mathrm{0xabcd}) = 0$ $\quad$ $\mathit{pf}(\mathrm{0x1000}) = \mathrm{0x20ab58bbaa53a22ad7}$

✓ ✗ ?

Point Functions

Partial point functions for key selection

$$\begin{array}{rcccc} f(x,y,c,k) := & & (n \mod k == 0) & \cdot & x+y \\ & + & \mathit{pf}(k) & \cdot & x-y << c \end{array}$$

$$\mathit{pf}(k) := ((0\mathrm{xff} \wedge k) \oplus 0\mathrm{xcd}) \cdot 0\mathrm{x}28\mathrm{cbfbeb}9\mathrm{a}020\mathrm{a}33$$

$\mathit{pf}(0\mathrm{x}1336) = 1$ $\quad$ $\mathit{pf}(0\mathrm{xabcd}) = 0$ $\quad$ $\mathit{pf}(0\mathrm{x}1000) = 0\mathrm{x}20\mathrm{ab}58\mathrm{bbaa}53\mathrm{a}22\mathrm{ad}7$

$\mathit{pf}(0\mathrm{xdead}) = 0\mathrm{xf}4\mathrm{c}7\mathrm{e}7859\mathrm{c}0\mathrm{c}3\mathrm{d}320$

✓ ✗ ??

Point Functions

Partial point functions for key selection

$$\begin{aligned} f(x, y, c, k) := \quad & (n \mod k == 0) && \cdot \quad x + y \\ + \quad & pf(k) && \cdot \quad x - y << c \end{aligned}$$

Point functions subvert I/O sampling

$pf(0\text{x}1336) = 1$

✓

$pf(0\text{xabcd}) = 0$

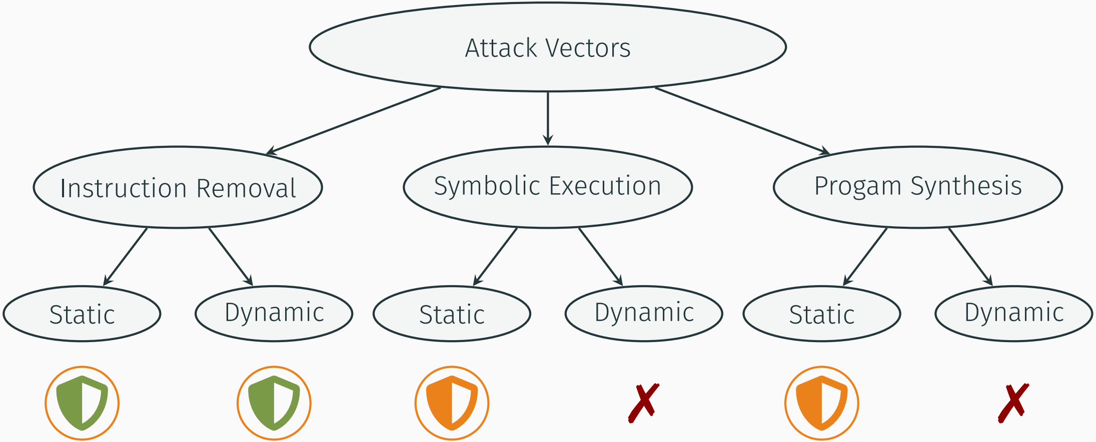
✗

$pf(0\text{x}1000) = 0\text{x}20\text{ab}58\text{bbaa}53\text{a}22\text{ad}7$

$pf(0\text{xdead}) = 0\text{xf}4\text{c}7\text{e}7859\text{c}0\text{c}3\text{d}320$

??

SMT-hard Key Encodings and Point Functions


Attack Vectors

Instruction Removal

Symbolic Execution

Progam Synthesis

Static

Dynamic

Static

Dynamic

Static

Dynamic

🛡 🛡 🛡 ✗ 🛡 ✗

Thwarting Program Synthesis

__v_add
__v_mul
__v_add
__v_add

⟶

__v_add_mul_add_add

Thwarting Program Synthesis

$$\begin{aligned} f(x, y, c, k) := \quad & (n_1 \mod k == 0) \cdot x + y \\ + \quad & pf(k) \cdot x - y << c \end{aligned}$$

Thwarting Program Synthesis

$$
\begin{aligned}
f(x, y, c, k) := \quad & (n_1 \mod k == 0) \cdot x + y + (x + x) \\
+ \quad & pf(k) \cdot x - y \cdot (x + y)
\end{aligned}
$$

Thwarting Program Synthesis

$$\begin{aligned} f(x, y, c, k) :=\ & (n_1 \bmod k == 0) \cdot \overbrace{x + y + (x + x)}^{\textit{target specific}} \\ +\ & pf(k) \cdot x - y \cdot (x + y) \end{aligned}$$

Thwarting Program Synthesis

target specific

Semantically complex arithmetic operations

How to Build Semantically Complex Operations

`mov edx, eax`	`edx.1 := eax`
`mov ecx, 0x20`	`ecx.1 := 0x20`
`add edx, ecx`	`edx.2 := edx.1 + ecx.1`
`imul edx, 0x10`	`edx.3 := edx.2 * 0x10`

How to Build Semantically Complex Operations

```
mov edx, eax
mov ecx, 0x20
add edx, ecx
imul edx, 0x10
```

```
edx.1 := eax
ecx.1 := 0x20
edx.2 := edx.1 + ecx.1
edx.3 := edx.2 * 0x10
```

Recursively replace uses by their definitions

How to Build Semantically Complex Operations

`mov edx, eax`	`edx.1 := eax`
`mov ecx, 0x20`	`ecx.1 := 0x20`
`add edx, ecx`	`edx.2 := edx.1 + ecx.1`
`imul edx, 0x10`	`edx.3 := edx.2 * 0x10`

Recursively replace uses by their definitions

How to Build Semantically Complex Operations

```
mov edx, eax
mov ecx, 0x20
add edx, ecx
imul edx, 0x10
```

```
edx.1 := eax
ecx.1 := 0x20
edx.2 := edx.1 + ecx.1
edx.3 := edx.2 * 0x10
```

Recursively replace uses by their definitions

How to Build Semantically Complex Operations

```
mov edx, eax
mov ecx, 0x20
add edx, ecx
imul edx, 0x10
```

```
edx.1 := eax
ecx.1 := 0x20
edx.2 := edx.1 + ecx.1
edx.3 := edx.2 * 0x10
```

Recursively replace uses by their definitions

```
edx.3 := edx.2 * 0x10
```

How to Build Semantically Complex Operations

```
mov edx, eax
mov ecx, 0x20
add edx, ecx
imul edx, 0x10
```

```
edx.1 := eax
ecx.1 := 0x20
edx.2 := edx.1 + ecx.1
edx.3 := edx.2 * 0x10
```

Recursively replace uses by their definitions

```
edx.3 := edx.2 * 0x10
```

How to Build Semantically Complex Operations

`mov edx, eax`	`edx.1 := eax`
`mov ecx, 0x20`	`ecx.1 := 0x20`
`add edx, ecx`	`edx.2 := edx.1 + ecx.1`
`imul edx, 0x10`	`edx.3 := edx.2 * 0x10`

Recursively replace uses by their definitions

```
edx.3 := edx.2 * 0x10 = (edx.1 + ecx.1) * 0x10
```

How to Build Semantically Complex Operations

```
mov edx, eax
mov ecx, 0x20
add edx, ecx
imul ed
```

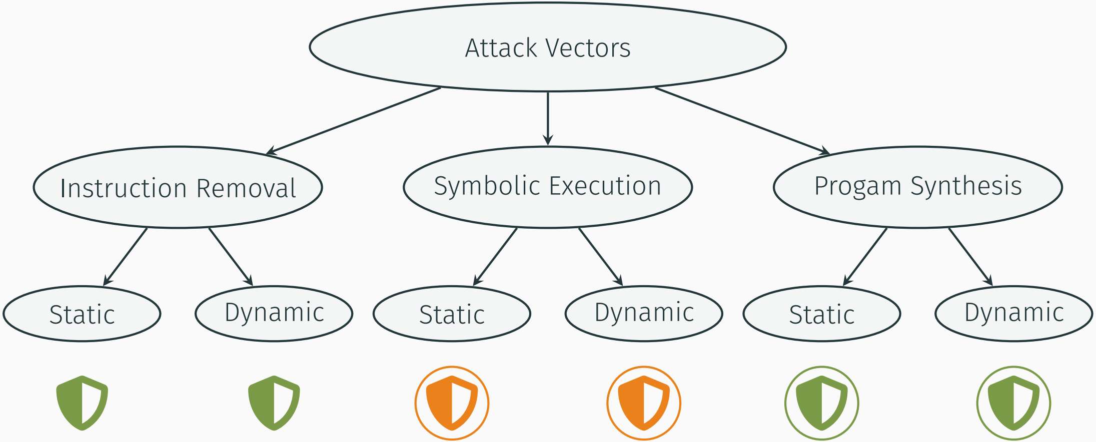
```
edx.1 := eax
ecx.1 := 0x20
edx.2 := edx.1 + ecx.1
* 0x10
```

$$f(x, y, c) := (x + y) * c$$

Recursively replace uses by their definitions

```
edx.3 := edx.2 * 0x10 = (edx.1 + ecx.1) * 0x10
```

Semantically Complex Operations



Thwarting Symbolic Execution

$$\begin{aligned} f(x, y, c, k) := \quad & (n_1 \mod k == 0) \quad \cdot \quad x + y + (x + x) \\ + \quad & pf(k) \quad \cdot \quad x - y \cdot (x + y) \end{aligned}$$

Thwarting Symbolic Execution

$$\begin{aligned} f(x,y,c,k) := \quad & (n_1 \mod k == 0) && \cdot \quad ((x \oplus y) + 2 \cdot (x \wedge y)) + (x \ll 1) \\ + \quad & pf(k) && \cdot \quad x - y \cdot (x + y) \end{aligned}$$

Thwarting Symbolic Execution

$$\begin{aligned} f(x, y, c, k) := \quad & (n_1 \mod k == 0) && \cdot && ((x \oplus y) + 2 \cdot (x \wedge y)) + (x \ll 1) \\ + \quad & pf(k) && \cdot && (x + \neg y + 1) \cdot ((x \oplus y) + 2 \cdot (x \wedge y)) \end{aligned}$$

Thwarting Symbolic Execution

$f(x, y,$ … $x \ll 1)$

… $2 \cdot (x \wedge y))$

Syntactically complex expressions

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

47) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

47) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

47) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

$$\begin{aligned}
&1) \quad x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y) \\
&2) \quad x \oplus y \rightarrow (x \vee y) - (x \wedge y) \\
&\qquad \dots \\
&47) \quad x \wedge y \rightarrow (\neg x \vee y) - \neg x
\end{aligned}$$

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))$$

Mixed Boolean Arithmetic Expressions

$$\begin{tikzcd}
x - y \cdot (x + y) \arrow[d, Leftrightarrow, "equal"] \\
x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))
\end{tikzcd}$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\ldots$

47) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

47) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

$$\boxed{x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))}$$

final expression

Traditional Approach

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\ldots$

(47) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))$$

final expression

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

847,000) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))$$

final expression

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

Lookup table w/ *all* identities

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))$$

final expression

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

$$\begin{aligned} 1)&\quad x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y) \\ 2)&\quad x \oplus y \rightarrow (x \vee y) - (x \wedge y) \\ &\quad \dots \\ 847{,}000)&\quad x \wedge y \rightarrow (\neg x \vee y) - \neg x \end{aligned}$$

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))$$

~~*final expression*~~

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (x + y)$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

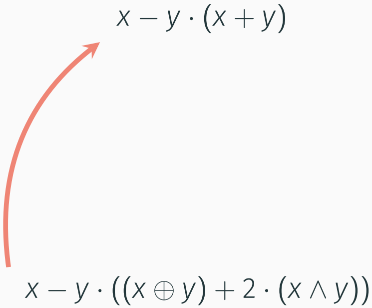
847,000) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))$$

~~*final expression*~~

Recursive Approach

Mixed Boolean Arithmetic Expressions



$$x - y \cdot (x + y)$$

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

847,000) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

Recursive Approach

Mixed Boolean Arithmetic Expressions

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \land y))$$

Rewriting rules:

$$\begin{array}{rl} 1) & x + y \rightarrow (x \oplus y) + 2 \cdot (x \land y) \\ 2) & x \oplus y \rightarrow (x \lor y) - (x \land y) \\ & \dots \\ 847{,}000) & x \land y \rightarrow (\neg x \lor y) - \neg x \end{array}$$

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \land y))$$

Recursive Approach

Mixed Boolean Arithmetic Expressions

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

847,000) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y))$$

Recursive Approach

Mixed Boolean Arithmetic Expressions

$$x - y \cdot ((x \oplus y) + 2 \cdot (x \land y))$$

Rewriting rules:

$$\begin{array}{rl} 1) & x + y \rightarrow (x \oplus y) + 2 \cdot (x \land y) \\ 2) & x \oplus y \rightarrow (x \lor y) - (x \land y) \\ & \ldots \\ 847{,}000) & x \land y \rightarrow (\neg x \lor y) - \neg x \end{array}$$

$$x - y \cdot (((x \lor y) - (x \land y)) + 2 \cdot (x \land y))$$

Recursive Approach

Mixed Boolean Arithmetic Expressions

$$\begin{tikzcd}
x - y \cdot ((x \oplus y) + 2 \cdot (x \wedge y)) \arrow[d, Leftrightarrow, "equal"] \\
x - y \cdot (((x \vee y) - (x \wedge y)) + 2 \cdot (x \wedge y))
\end{tikzcd}$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$
$\dots$
847,000) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

Recursive Approach

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (((x \vee y) - (x \wedge y)) + 2 \cdot (x \wedge y))$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$
$\dots$
847,000) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

$$x - y \cdot (((x \vee y) - (x \wedge y)) + 2 \cdot (x \wedge y))$$

Recursive Approach

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (((x \vee y) - (x \wedge y)) + 2 \cdot (x \wedge y))$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

847,000) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

$$x - y \cdot (((x \vee y) - (x \wedge y)) + 2 \cdot (x \wedge y))$$

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (((x \vee y) - (x \wedge y)) + 2 \cdot (x \wedge y))$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$

2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

$\dots$

847,000) $x \wedge y \rightarrow (\neg x \vee y) - \neg x$

$$x - y \cdot (((x \vee y) - ((\neg x \vee y) - \neg x)) + 2 \cdot (x \wedge y))$$

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (((x \lor y) - (x \land y)) + 2 \cdot (x \land y))$$

Rewriting rules:

$$\begin{aligned} &1) && x + y \rightarrow (x \oplus y) + 2 \cdot (x \land y) \\ &2) && x \oplus y \rightarrow (x \lor y) - (x \land y) \\ & && \ldots \\ &847{,}000) && x \land y \rightarrow (\neg x \lor y) - \neg x \end{aligned}$$

$$\boxed{x - y \cdot (((x \lor y) - ((\neg x \lor y) - \neg x)) + 2 \cdot (x \land y))}$$

final expression

Mixed Boolean Arithmetic Expressions

$$x - y \cdot (((x \vee y) - (x \wedge y)) + 2 \cdot (x \wedge y))$$

Rewriting rules:

1) $x + y \rightarrow (x \oplus y) + 2 \cdot (x \wedge y)$
2) $x \oplus y \rightarrow (x \vee y) - (x \wedge y)$

Recursive Rewriting

$\neg x \vee y) - \neg x$

$$x - y \cdot (((x \vee y) - ((\neg x \vee y) - \neg x)) + 2 \cdot (x \wedge y))$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Invertible function on 1 byte:

$$\begin{aligned} h: &\quad a \mapsto 39a + 23 \\ h^{-1}: &\quad a \mapsto 151a + 111 \end{aligned}$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Invertible function on 1 byte:

$$\begin{aligned} h: \quad & a \mapsto 39a + 23 \\ h^{-1}: \quad & a \mapsto 151a + 111 \end{aligned}$$

$$\implies expr \equiv h^{-1}(h(expr)) \mod 2^8$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Invertible function on 1 byte:

$$h : \quad a \mapsto 39a + 23$$
$$h^{-1} : \quad a \mapsto 151a + 111$$

$$\implies expr \equiv h^{-1}(h(expr)) \mod 2^8$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

$$x - y \cdot (h^{-1}(h(x + y)))$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Invertible function on 1 byte:

$$\begin{aligned} h :&\quad a \mapsto 39a + 23 \\ h^{-1} :&\quad a \mapsto 151a + 111 \end{aligned}$$

$$\implies expr \equiv h^{-1}(h(expr)) \mod 2^8$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

$$x - y \cdot (h^{-1}(h(x + y)))$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Invertible function on 1 byte:

$$\begin{aligned} h &: \quad a \mapsto 39a + 23 \\ h^{-1} &: \quad a \mapsto 151a + 111 \end{aligned}$$

$$\implies expr \equiv h^{-1}(h(expr)) \mod 2^8$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

$$x - y \cdot (h^{-1}(h(x + y)))$$

$$x - y \cdot (h^{-1}(39 \cdot (x + y) + 23))$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Invertible function on 1 byte:

$$h: \quad a \mapsto 39a + 23$$

$$h^{-1}: \quad a \mapsto 151a + 111$$

$$\implies expr \equiv h^{-1}(h(expr)) \mod 2^8$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

$$x - y \cdot (h^{-1}(h(x + y)))$$

$$x - y \cdot (h^{-1}(39 \cdot (x + y) + 23)) \longrightarrow$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Invertible function on 1 byte:

$$h : \quad a \mapsto 39a + 23$$

$$h^{-1} : \quad a \mapsto 151a + 111$$

$$\Longrightarrow \quad expr \equiv h^{-1}(h(expr)) \mod 2^8$$

Polynomial MBAs

$$x - y \cdot (x + y)$$

$$x - y \cdot (h^{-1}(h(x + y)))$$

$$x - y \cdot (h^{-1}(39 \cdot (x + y) + 23))$$

$$x - y \cdot (151 \cdot (39 \cdot (x + y) + 23) + 111)$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Invertible function on 1 byte:

$$h : \quad a \mapsto 39a + 23$$

$$h^{-1} : \quad a \mapsto 151a + 111$$

$$\Longrightarrow \; expr \equiv h^{-1}(h(expr)) \mod 2^8$$

Polynomial MBAs

$$\begin{tikzcd} x - y \cdot (x + y) \arrow[d, Leftrightarrow, "\mathit{equal}"] \\ x - y \cdot (151 \cdot (39 \cdot (x + y) + 23) + 111) \end{tikzcd}$$

Rewrite as:

$$expr \equiv h^{-1}(h(expr))$$

Invertible function on 1 byte:

$$\begin{aligned} h : &\quad a \mapsto 39a + 23 \\ h^{-1} : &\quad a \mapsto 151a + 111 \end{aligned}$$

$$\implies expr \equiv h^{-1}(h(expr)) \mod 2^8$$

Binary Permutation Polynomial Inversion and Application to Obfuscation Techniques

Lucas Barthelemy[abd]
lbarthelemy@quarkslab.com

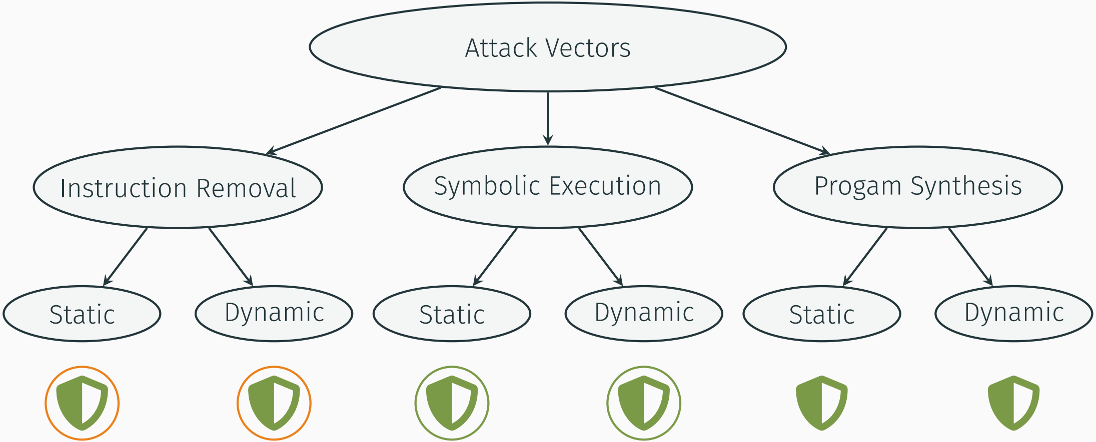
Ninon Eyrolles[a]
neyrolles@quarkslab.com

Guenaël Renault[bce]
guenael.renault@upmc.fr

Raphaël Roblin[bd]
raph.roblin@gmail.com

[a]Quarkslab, Paris, France
[b]Sorbonne Universités, UPMC Univ Paris 06, F-75005, Paris, France
[c]CNRS, UMR 7606, LIP6, F-75005, Paris, France
[d]UPMC Computer Science Master Department, SFPN Course
[e]Inria, Paris Center, PolSys Project

Syntactically Complex Operations



Attack Vectors

Instruction Removal

Symbolic Execution

Progam Synthesis

Static

Dynamic

Static

Dynamic

Static

Dynamic

Taking it all together

Loki: Academic Next-Gen VM Prototype

Design Principle #1 – Complex and target-specific instruction sets.

Design Principle #2 – Intertwining VM components.

Loki: Academic Next-Gen VM Prototype

Design Principle #1 – Complex and target-specific instruction sets.

Design Principle #2 – Intertwining VM components.

- **merged semantics** to encforce **cross-handler** analysis

Loki: Academic Next-Gen VM Prototype

Design Principle #1 – Complex and target-specific instruction sets.

Design Principle #2 – Intertwining VM components.

- **merged semantics** to encforce **cross-handler** analysis
- **polynomial encodings** to **interlock** instruction semantics

Loki: Academic Next-Gen VM Prototype

Design Principle #1 – Complex and target-specific instruction sets.

Design Principle #2 – Intertwining VM components.

- **merged semantics** to encforce **cross-handler** analysis
- **polynomial encodings** to **interlock** instruction semantics
- **point functions** to **subvert I/O sampling**

Loki: Academic Next-Gen VM Prototype

Design Principle #1 – Complex and target-specific instruction sets.

Design Principle #2 – Intertwining VM components.

- **merged semantics** to encforce **cross-handler** analysis
- **polynomial encodings** to **interlock** instruction semantics
- **point functions** to **subvert I/O sampling**
- complex, **data-flow dependent** instruction semantics to thwart **program synthesis**

Loki: Academic Next-Gen VM Prototype

Design Principle #1 – Complex and target-specific instruction sets.

Design Principle #2 – Intertwining VM components.

- **merged semantics** to encforce **cross-handler** analysis
- **polynomial encodings** to **interlock** instruction semantics
- **point functions** to **subvert I/O sampling**
- complex, **data-flow dependent** instruction semantics to thwart **program synthesis**
- **MBAs** to thwart **symbolic execution**

Impact on Deobfuscation

Verging on the Limits

Challenges in Code Deobfuscation

Design Principle #1 – Complex and target-specific instruction sets.

- synthesis-based attacks are no longer feasible
- no **meaningful** instruction **mnemonics** for disassemblers

`vadd` vs. `vneg_vadd_vmul_vxor_vpush`

Challenges in Code Deobfuscation

Design Principle #2 – Intertwining VM components.

- shift towards **global analysis**; larger analysis scope required
- analysis **effort rises enormously**: limitations of binary analysis techniques & tools

What needs to be done?

Better Analysis Tools

- better support for **interprocedural** & **multi-threaded** analysis
- improve **tooling for large instruction sequences** (performance and memory footprint)
- advances in **binary lifting**

Yes, these are hard problems.

Selection of Analysis Windows

- **identification** of relevant **sources** and **sinks**
- strategies to **isolate** and **simplify** (partial) **data flows**
- automated **exploration** of **control** and **data flows** (CFG/DFG construction)

Advances in MBA Deobfuscation

- simplification of large **polynomial** MBAs
- improvements on **synthesis-based approaches** to reach higher semantic depths
- strategies to synthesize **constants**

$$(x \oplus 0xf5692443e29a24c2) \cdot 0x3886553866f35c17$$

Research Catches Up

Linear MBAs are dead

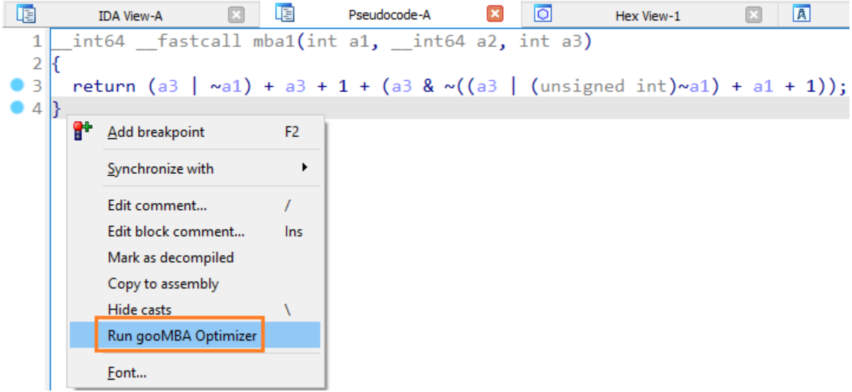
Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions

Benjamin Reichenwallner & Peter Meerwald-Stadler
Denuvo GmbH
Salzburg, Austria

---

https://github.com/DenuvoSoftwareSolutions/SiMBA

Hex-Rays Decompiler Plugin

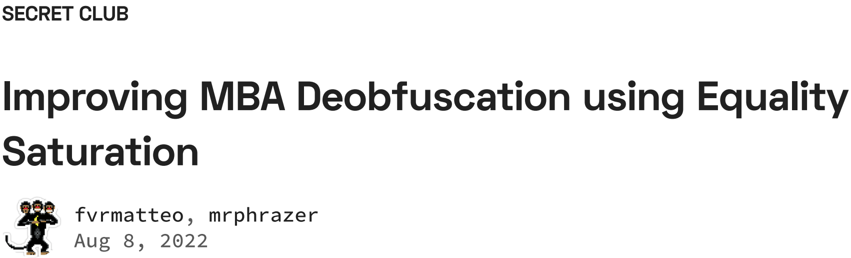


```
__int64 __fastcall mba1(int a1, __int64 a2, int a3)
{
  return (a3 | ~a1) + a3 + 1 + (a3 & ~((a3 | (unsigned int)~a1) + a1 + 1));
}
```

---

https://github.com/HexRaysSA/goomba

Novel Methods for Complex MBAs



SECRET CLUB

Improving MBA Deobfuscation using Equality Saturation

fvrmatteo, mrphrazer
Aug 8, 2022

`https://secret.club/2022/08/08/eqsat-oracle-synthesis.html`

However …

Open Challenges

- analysis tools still insufficient
- selection of analysis windows remains challenging
- low impact of MBA deobfuscation in practice

Open Challenges

- analysis tools still insufficient

- selection of analysis windows remains challenging

Deobfuscation still not feasible

- low impact of MBA deobfuscation in practice

Conclusion

Takeaways

1. current VMs can be broken in a (semi-)automated fashion
2. industry shifts to novel VM designs
3. code deobfuscation research has to catch up despite recent advancements

Takeaways

1. current VMs can be broken in a (semi-)automated fashion
2. industry shifts to novel VM designs
3. code deobfuscation research has to catch up despite recent advancements

Next-gen VMs will shape the landscape of modern obfuscation in the next years.

Summary

- virtualization-based obfuscation
- attacks on VMs (instruction removal, symbolic execution, program synthesis)
- next-gen VMs and their impact on deobfuscation
- recent advancements in MBA deobfuscation

Tim Blazytko
`@mr_phrazer`
`synthesis.to`

Moritz Schloegel
`@m_u00d8`
`mschloegel.me`