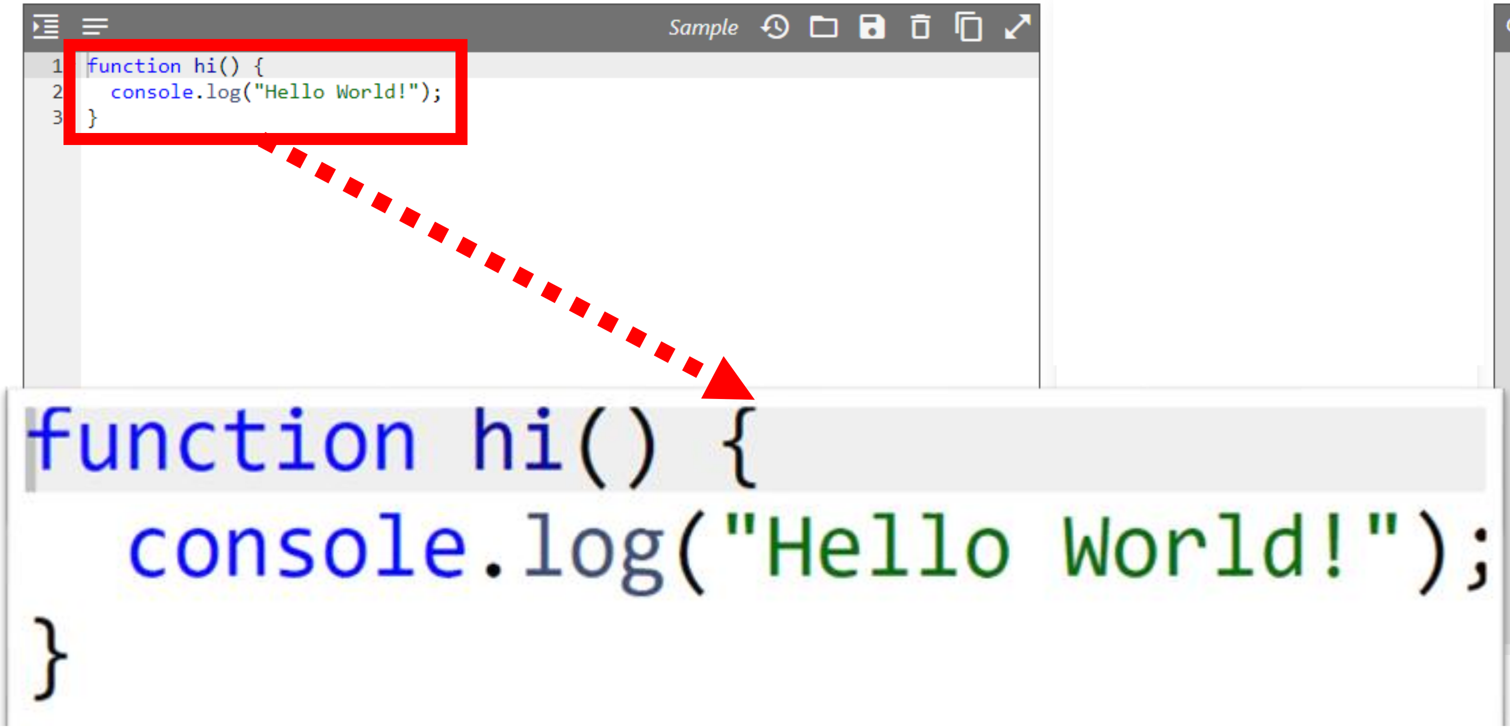
Analysis of virtual machine based obfuscators

Vladislav Hrčka
RNDr. Jaroslav Janáček, PhD
Mgr. Peter Košinár

Introduction – obfuscation

- Obfuscated javascript:



Aim

- describe virtual machine based obfuscators in general
- summarize methods to deal with such kind of obfuscators
- analyze a specific virtual machine
- design and implement methods to deobfuscate code protected with the specific machine

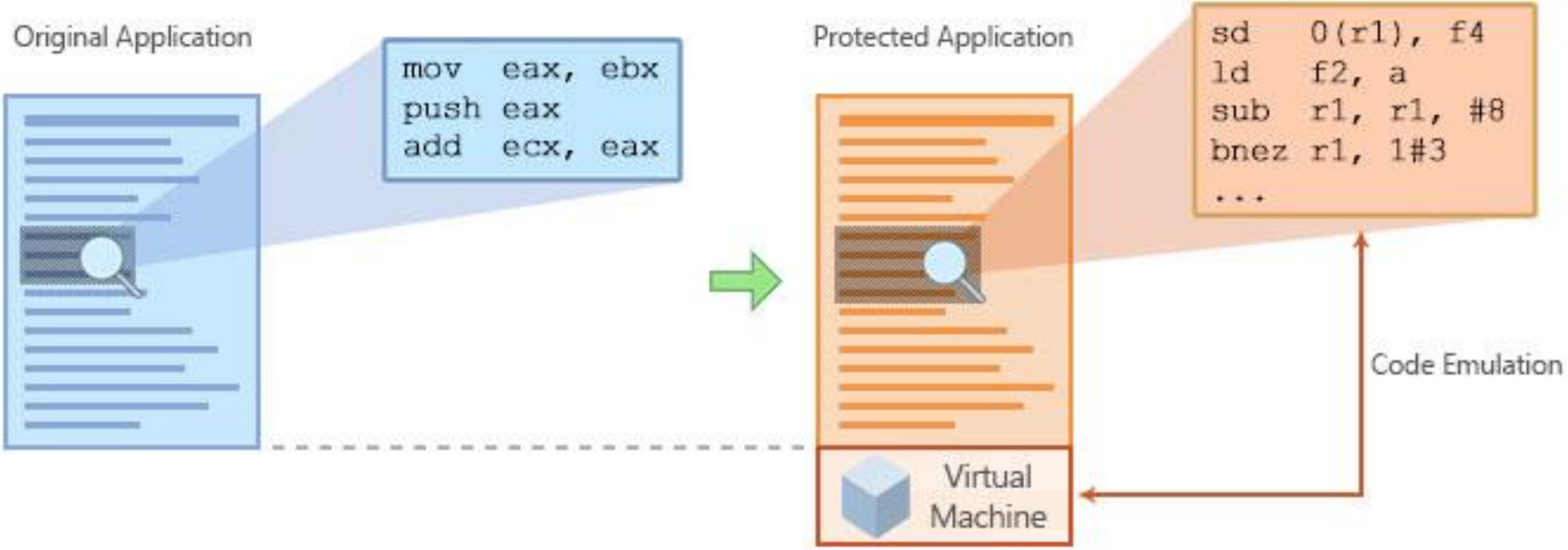
CodeVirtualizer

- *"Code Virtualizer is a powerful code obfuscation system for Windows applications that helps developers to protect their sensitive code areas against Reverse Engineering with very strong obfuscation code, based on code virtualization."*

```
#include <stdio.h>
#include "VirtualizerSDK.h"

void main()
{
    VIRTUALIZER_START      // the area to protect starts here

    printf("Hello World");

    VIRTUALIZER_END        // end of area to protect
}
```

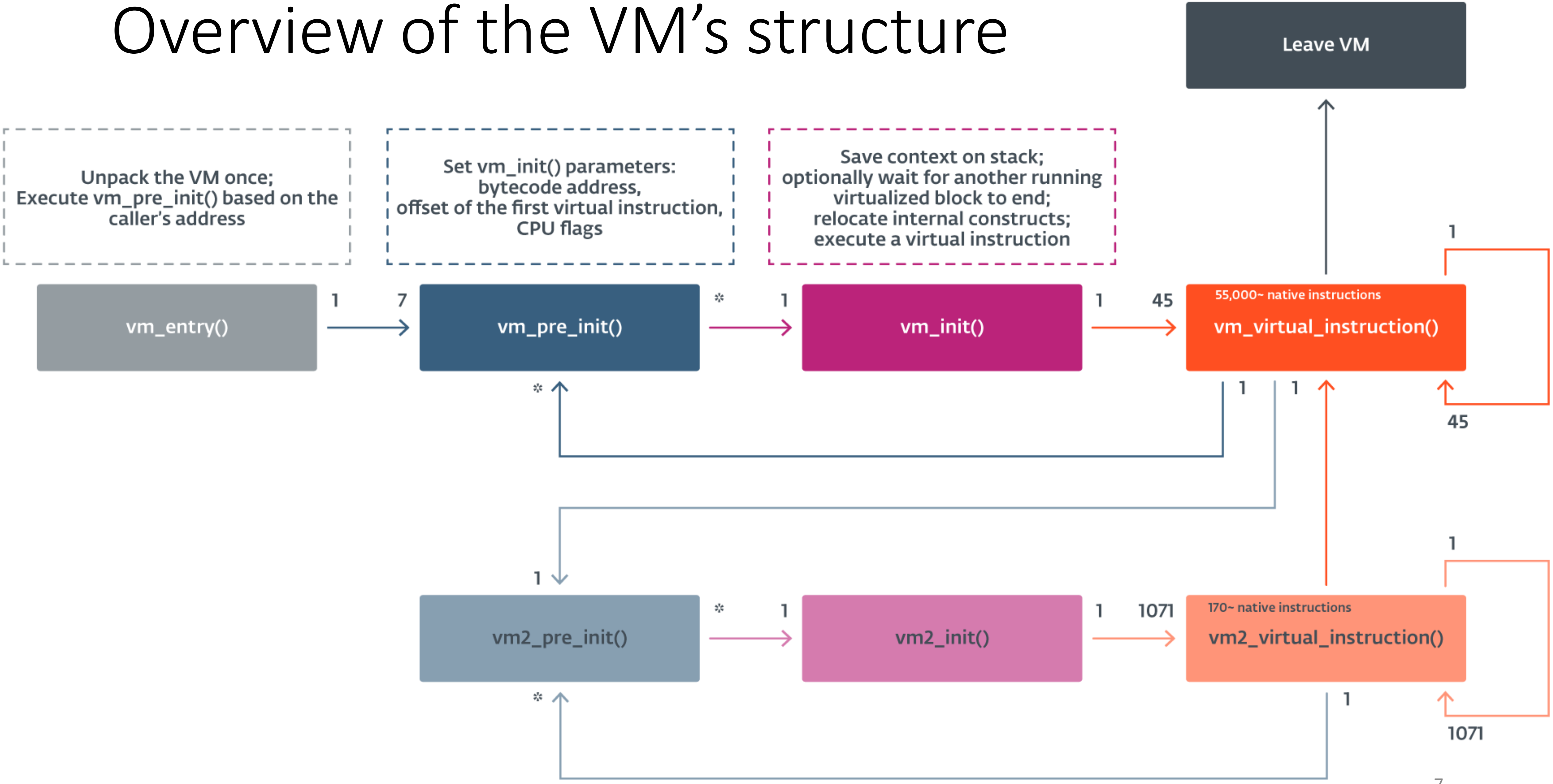
https://www.oreans.com/CodeVirtualizer.php

CodeVirtualizer



https://www.oreans.com/assets/images/figure_final.jpg

Overview of the VM's structure



Attacks

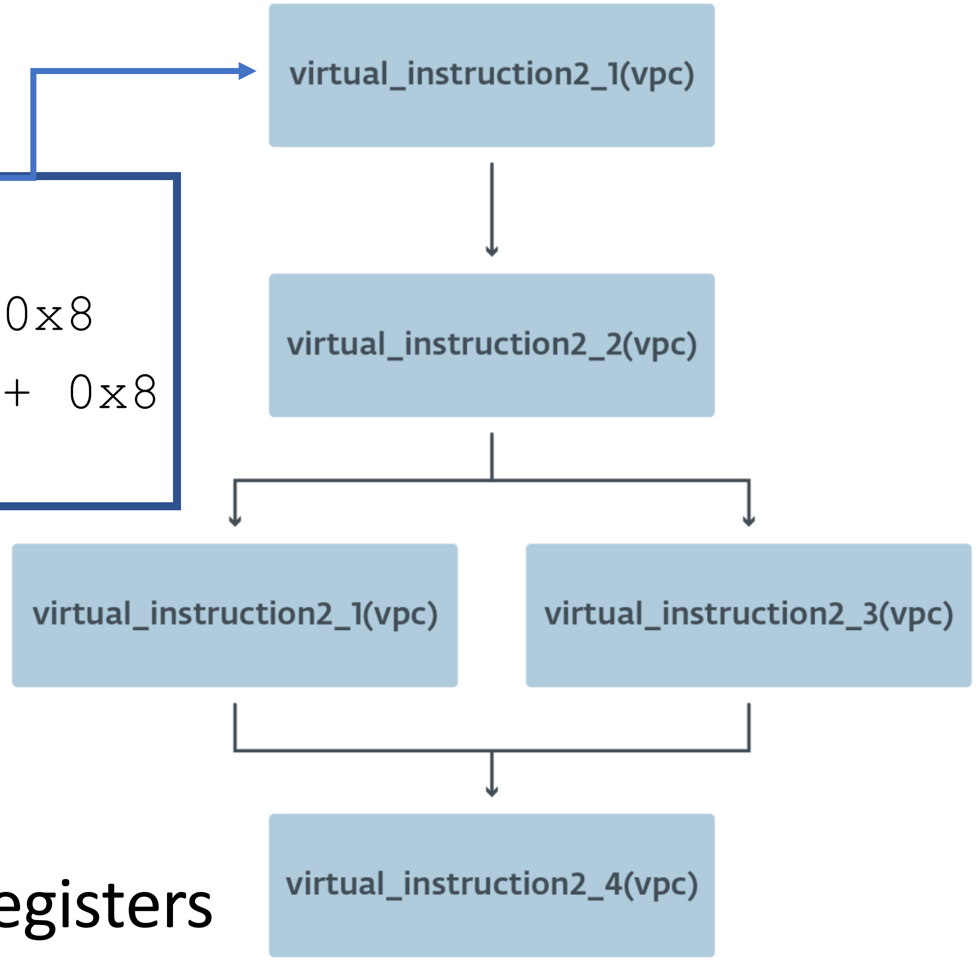
- Manually build a disassembler
  - Rolles R. Unpacking virtualization obfuscators, 3rd USENIX Workshop on Offensive Technologies, 2009
- Program synthesis of handlers – slow
  - Blazytko T, Contag M, Aschermann C, Holz T, Syntia: Synthesizing the semantics of obfuscated code, 26th USENIX Security Symposium, pages 643-659, 2017
- Path coverage and trace merging – limitations and slow
  - Salwan J, Bardin S, Potet ML, Symbolic deobfuscation: From virtualized code back to the original, International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 372-392, 2018

Our approach

- Virtual instruction summary:

```
RIP = @64[@64[RBP_init + 0xA4] + 0x5A8]
@64[RBP_init + 0x28] = @64[RBP_init + 0x28] + 0x8
@64[RBP_init + 0x141] = @64[RBP_init + 0x141] + 0x8
@64[RBP_init + 0x12A] = @64[RSP_init]
```

- Build a graph from summaries
- Treat some values as concrete:
    - Rolling decryption registers
    - Memory accesses relative to the bytecode pointer
- Between blocks preserve only the decryption registers



Virtual instruction summary

```
IRDst = ({(@16[RBP_init + 0xB] + -(@32[RBP_init + 0x70] ^ {@16[@64[RBP_i
@16[RBP_init + 0xB] = @16[RBP_init + 0xB] + -(@32[RBP_init + 0x70] ^ {@1
@32[RBP_init + 0x70] = @32[RBP_init + 0x70] & (@32[RBP_init + 0x70] ^ {@
@64[RBP_init + {(@16[RBP_init + 0xB] + -(@32[RBP_init + 0x70] ^ {@16[@64
```

```
RIP = @64[@64[RBP_init + 0xA4] + 0x5A8]
@64[RBP_init + 0x28] = @64[RBP_init + 0x28] + 0x8
@64[RBP_init + 0x141] = @64[RBP_init + 0x141] + 0x8
@64[RBP_init + 0x12A] = @64[RSP_init]
```

```
RIP = @64[instruction_table + 0x5A8]
IEIP = IEIP + 0x8
IRSP = IRSP + 0x8
REG1 = @64[RSP_init]
```

```
POP REG1
```

Source code init

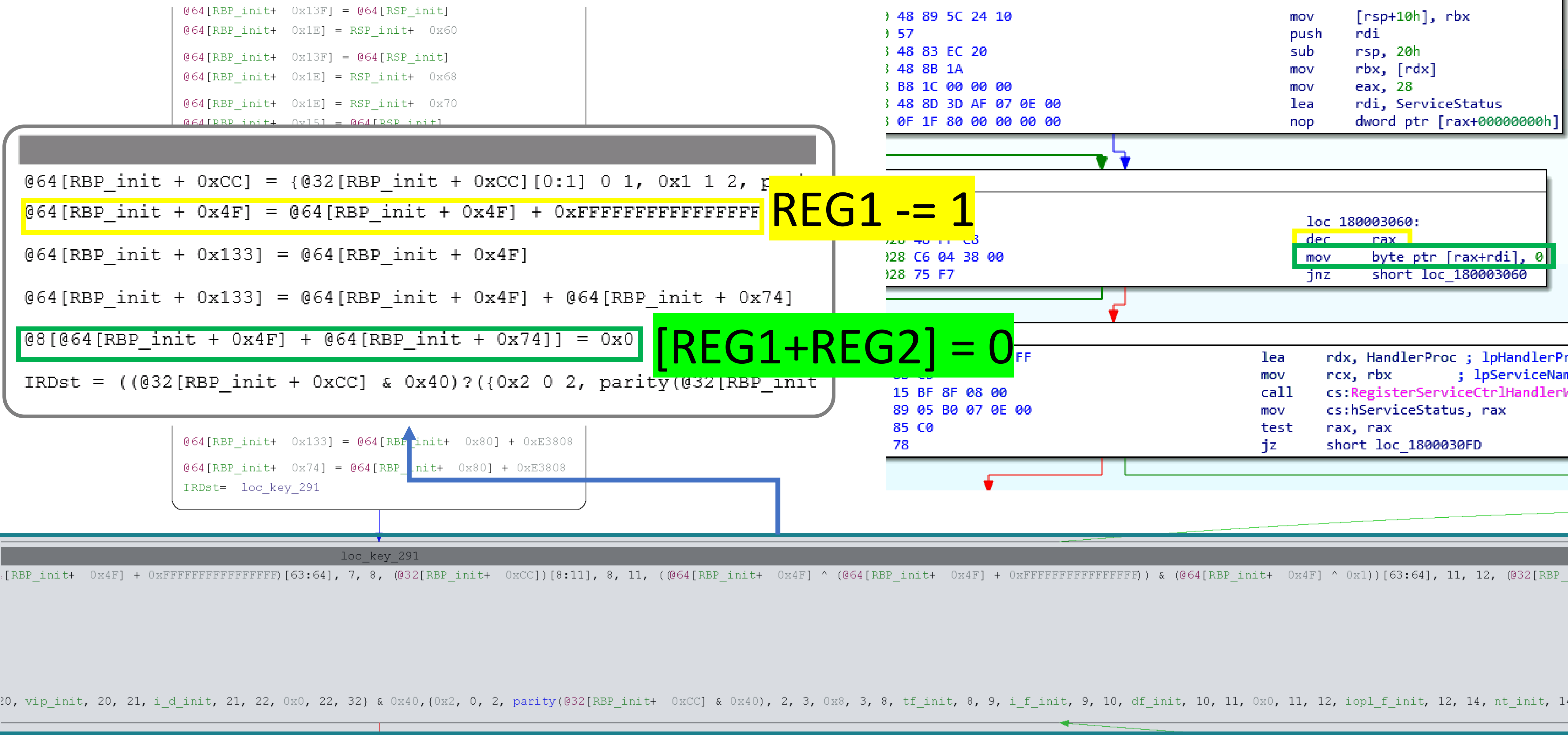
```
class VM1(Wslink):
    def __init__(self, file_path, reloc=0):
        super().__init__(file_path, reloc)
        self.instr_table_addr = 0x11de70
        self.obf_reg_offsets = {(0x8, 8), (0x11, 32), (0x26, 32), (0x60, 16), (0xbe, 32),
                                (0xd4, 32), (0x13b, 32), (0x147, 32), (0x107, 64),
                                (0xc2, 64), (0xae, 64), (0x3e, 8), (0x34, 8), (0x102, 8),
                                (0x35, 8), (0x3f, 64), (0x92, 64), (0x57, 8),
                                (0x1d, 8)}
        self.vm_base = 0xf9206
        self.vm_pc_off = 0x2c
        self.instr_table_off = 0xee


reloc = -0xf33f5
vma = VM1('extracted_vm.dmp', reloc)
first_executed_virt_instr = 0x21dbea + reloc
first_virt_instr_in_table = 0x21B4E0 + reloc
vma.process(first_executed_virt_instr, 0x19, opt=True)
```

DEOBFUSCATED

```
@64[RBP_init+  0x15] = @64[RSP_init]

@64[RBP_init+  0x11F] = @64[RSP_init]
@64[RBP_init+  0x1E] = RSP_init+  0x78

@64[RBP_init+  0x4F] = @64[RSP_init]
@64[RBP_init+  0x1E] = RSP_init+  0x80

@64[RBP_init+  0xCC] = @64[RSP_init]
@64[RBP_init+  0x1E] = RSP_init+  0x88

@64[RBP_init+  0x1E] = RSP_init+  0x98

@64[RBP_init+  0x13F] = @64[@64[RBP_init+  0x13F]]

@32[RBP_init+  0x53] = 0x0
@32[RBP_init+  0x4F] = 0x1C

@64[RBP_init+  0x133] = 0x3092

@64[RBP_init+  0x133] = @64[RBP_init+  0x80] + 0x3092

@64[RBP_init+  0x133] = @64[RBP_init+  0x80] + 0xE3808

@64[RBP_init+  0x74] = @64[RBP_init+  0x80] + 0xE3808
IRDst=  loc_key_291
```

```
loc_key_291
== 0x1, 6, 7, (@64[RBP_init+  0x4F] + 0xFFFFFFFFFFFFFFFF)[63:64], 7, 8, (@32[RBP_init+  0xCC])[8:11], 8, 11, ((@64[RBP_init+  0x4F] ^ (@64[RBP_init+  0x4F] + 0xFFFFFFFFFFFFFFFF)) & (@64[RBP_init+  0x4F] ^ 0x1))[63:64], 11, 12, (@32[RBP_init+  0xCC])[12:15], 12, 15, 0x0, 15, 16, (@

9, vif_init, 19, 20, vip_init, 20, 21, i_d_init, 21, 22, 0x0, 22, 32} & 0x40,{0x2, 0, 2, parity(@32[RBP_init+  0xCC] & 0x40), 2, 3, 0x8, 3, 8, tf_init, 8, 9, i_f_init, 9, 10, df_init, 10, 11, 0x0, 11, 12, iopl_f_init, 12, 14, nt_init, 14, 15, 0x0, 15, 16, rf_init, 16, 17, vm_init,
```

```
loc_key_318
@64[RBP_init+  0x133] = 0x30A2

@64[RBP_init+  0x133] = @64[RBP_init+  0x80] + 0x30A2

@64[RBP_init+  0x133] = @64[RBP_init+  0x80] + 0x2FB0

@64[RBP_init+  0x15] = @64[RBP_init+  0x80] + 0x2FB0

@64[RBP_init+  0x11F] = @64[RBP_init+  0x13F]

@64[RBP_init+  0x133] = 0x30AB

@64[RBP_init+  0x133] = @64[RBP_init+  0x80] + 0x30AB

@64[RBP_init+  0x133] = @64[RBP_init+  0x80] + 0x8C038

@64[RSP_init+  0xFFFFFFFFFFFFFFF8] = @64[RBP_init+  0x4F]
@64[RBP_init+  0x1E] = @64[RBP_init+  0x1E] + 0xFFFFFFFFFFFFFFF8

@64[RSP_init+  0xFFFFFFFFFFFFFFF8] = @64[RBP_init+  0x4F]
@64[RBP_init+  0x1E] = @64[RBP_init+  0x1E] + 0xFFFFFFFFFFFFFFF8

@64[RSP_init+  0xFFFFFFFFFFFFFFF8] = @64[RBP_init+  0xCC]
@64[RBP_init+  0x1E] = @64[RBP_init+  0x1E] + 0xFFFFFFFFFFFFFFF8

@64[RSP_init+  0xFFFFFFFFFFFFFFF8] = @64[RBP_init+  0x4F]
@64[RBP_init+  0x1E] = @64[RBP_init+  0x1E] + 0xFFFFFFFFFFFFFFF8

@64[RSP_init+  0xFFFFFFFFFFFFFFF8] = @64[RBP_init+  0x11F]
@64[RBP_init+  0x1E] = @64[RBP_init+  0x1E] + 0xFFFFFFFFFFFFFFF8

@64[RSP_init+  0xFFFFFFFFFFFFFFF8] = @64[RBP_init+  0x15]
```



DEOBFUSCATED

ORIGINAL SAMPLE

Analyzing results

Bytecode block	VM1	VM2
Size in bytes	695	1,145
Total number of processed virtual instructions	62	109
Total number of underlying native instructions	3,536,427	17,406
Total number of resulting IR instructions (including IRDsts)	192	307
Execution time in seconds	382	10

Thank you