Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization

Mingyue Liang[1], Zhoujun Li[1(✉)], Qiang Zeng[2], and Zhejun Fang[3]

[1] Beihang University, Beijing, China
{liangmy,lizj}@buaa.edu.cn
[2] Temple University, Philadelphia, PA, USA
qzeng@temple.edu
[3] CNCERT/CC, Beijing, China
fzj@cert.org.cn

Abstract. Virtualization-obfuscation replaces native code in a binary with semantically equivalent and self-defined bytecode, which, upon execution, is interpreted by a custom virtual machine. It makes the code very difficult to analyze and is thus widely used in malware. How to deobfuscate such virtualization obfuscated code has been an important and challenging problem. We approach the problem from an innovative perspective by transforming it into a compilation optimization problem, and propose a novel technique that combines trace analysis, symbolic execution and compilation optimization to defeat virtualization obfuscation. We implement a prototype system and evaluate it against popular virtualization obfuscators; the results demonstrate that our method is effective in deobfuscation of virtualization-obfuscated code.

Keywords: Deobfuscation · Virtualization obfuscation
Symbolic execution · Compilation optimization

1 Introduction

Virtualization-based obfuscation replaces the code in a binary with semantically equivalent bytecode, which can only be interpreted by a virtual machine whose instruction set and architecture can be customized. Thus, it makes the resulting code difficult to understand and analyze, and is widely used in malware [1]. When regular dynamic and static analyzers are directly applied to analyzing such code, their execution gets trapped into the VM interpreter and thus can hardly reveal the real logic of the code. Therefore, how to deobfuscate virtualization-obfuscated code has been an important and challenging problem.

Existing techniques either reverse engineer the virtual machine to infer the logic behind the bytecode [2], or execute the obfuscated code and work on the instruction traces corresponding to the executed bytecode [3–5]. While the former relies on a complete reverse engineering of the virtual machine, which is challenging in itself, the latter requires advanced control/data-flow analysis (which usually involves many false negatives and positives and requires many *ad hoc* methods for handling different obfuscations) to remove redundant code. Indeed, due to garbage code insertion, one of the main challenges to deobfuscate virtualization-obfuscated code is how to correctly remove unneeded code and generate concise code. This challenge is not well resolved yet.

Our insight is that redundant code elimination is well resolved and implemented in the field of compilation optimization; thus, if we can leverage the power of compilers, the challenge above can be resolved elegantly. Thus, we approach the challenge from an innovative compilation-optimization perspective. Specifically, we first apply symbolic execution to summarize the semantics of each bytecode handler (the execution of each bytecode corresponds to the invocation of its handler), and then automatically transform the semantics of a handler into a function represented in some high-level programming language. Next, after the execution of bytecode is transformed into a piece of source code that represents the invocations of the corresponding functions, we leverage a compiler to compile the source code. Consequently, the compiler eliminates unneeded code of VM, such as operations on virtual stack and registers, thanks to compilation optimization, and generates deobfuscated code, which then can be analyzed using various off-the-shelf tools.

We made the following contributions.

(a) We propose an innovative idea that resolves the deobfuscation challenge from the perspective of compilation optimization, which makes it possible to reuse powerful code optimizations implemented in modern compilers.
(b) We propose a symbolic execution based method for automatically extracting semantic information of bytecode handlers, and represent the handlers in high-level programming languages.
(c) We have implemented a prototype system and evaluated it against representative obfuscators including VMProtect [6] and Code Virtualizer [7].

2 Background and Challenges

This section describes the background about virtualization obfuscation and discusses the challenges in the process of deobfuscation and code recovery.

2.1 Background: Virtualization Obfuscation

A virtualization obfuscator takes a binary file as input, parses its machine code instructions, and translates them to self-defined bytecode, which can be interpreted by an embedded interpreter during execution. Thus, a virtualization obfuscator has to implement a complete virtual machine that contains the virtual instruction set definition and a corresponding bytecode interpreter. Below, we describe some important concepts and details for virtualization obfuscation.

Virtual Machine. A *virtual machine*, also called an *emulator* or *interpreter*, is the core of an obfuscation system for interpreting bytecode instructions. As the original machine code is removed and replaced by virtual instructions, the obfuscated binary embeds a corresponding interpreter in the code section. A *Virtual Program Counter* in a VM works like the EIP register in x86 architecture; it stores the address pointing to the location of the current virtual instruction (described below). When interpreting virtual instructions, the virtual machine fetches the instruction pointed to by the VPC and execute it, after which VPC will be updated to point to the next instruction.

Virtual Instructions. Virtual instructions are also called *bytecode* instructions. Virtual instructions are translated from original machine code and encoded in the data section of the obfuscated binary. The virtual machine fetches bytecode as read-only data and interprets it with embedded *hander* functions. The execution of a virtual instruction corresponds to an invocation of the corresponding handler function. As the virtual instruction set is privately defined by the obfuscator and is quite different from public architectures such as Intel, ARM and MIPS, regular static analysis tools cannot analyze virtualization obfuscated code.

Virtual Registers. A virtual machine uses virtual registers to store temporary variable values, but they do not exactly match the x86/x64 general-purpose registers. For example, VMProtect VM has 16 virtual registers, which are randomly mapped to 8 Intel x86 registers to increase obfuscation.

Virtual Stack. Stack-based VMs are popular in virtualization obfuscation, and is also the target architecture of our work. In a stack-based VM, all data passing operations go through a virtual stack, and registers and memory never exchange data directly. Given an X86 instruction *add eax, ebx*, its equivalent virtual instruction sequence in a stack-based VM is:

```
vPush vR0
vPush vR1
vAdd
vPop vFlag
vPop vR2
```

The first double *vPush* instructions push two registers onto a virtual stack, after which the *vAdd* pops two variables from the stack to perform the addition and then stores the result and the side-effect flag back to the stack. The last *vPop* instructions pop the addition result and the flag to registers.

2.2 Challenges of Deobfuscation

While conventional runtime-packed code can be extracted from the memory when the code is executed and unpacked [8], virtualization-obfuscated code never restores the original code in the memory during execution. Thus, regular unpackers cannot recover the original code.

Another challenge is various bytecode-level obfuscation can be applied to the virtualization-obfuscated code, which makes the extracted bytecode even harder to analyze and understand. For example, a simple x86 instruction can be translated into several virtual instructions which keep same semantic but are much more complex to understand. A concrete example is logical operation obfuscation in VMProtect. A VM in VMProtect does not generate *not, and, or* or *xor* instructions but only nor instructions. All these logical operations are implemented in *nor* instructions; e.g., *or(a, b) = nor(nor(a, b), nor(a, b))*.

Various VM architecture of virtual obfuscators is also a challenge. Conventional deobfuscation tools works well on general architecture (Intel, ARM and MIPS) but do not support customized VM architecture without specialized adaptation. When deobfuscating virtualization-obfuscated code, there is a lack of a generic technique that tackles various VM architecture with different kinds of bytecode-level obfuscation. Our goal is to conquer this challenge and propose such a generic technique.

3 Deobfuscation Through Symbolic Execution and Compilation Optimization

3.1 Main Idea and Architecture

Instead of implementing various deobfuscation algorithms, we creatively propose to leverage the power of modern compilers, which perform advanced code optimization, to resolve the deobfuscation problem. However, it is infeasible to apply a regular compiler, such as gcc and clang, to processing custom virtual instructions directly. We thus propose to convert each virtual-instruction handler, which is relatively simple, to a function coded in some high-level programming language, and our approach is to summarize the semantics of handlers through symbolic execution. Subsequently, a sequence of virtual instructions can be represented as a sequence of invocations of these functions, which then can be processed by a regular compiler for automatic optimization to eliminate garbage code and obtain concise resulting code.

Figure 1 shows the architecture of our deobfuscation system, which comprises the following components: (1) A trace analysis module, which records runtime information of the obfuscated binary and does offline analysis to identify and

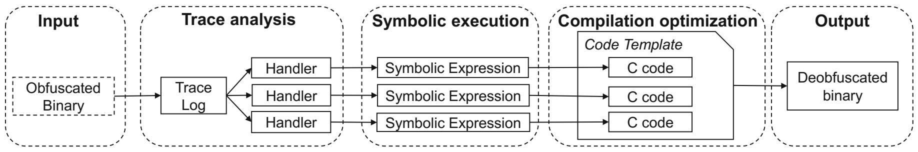

Fig. 1. Overview of our approach.

extract handler functions. (2) A symbolic execution module, which analyzes semantic information of each handler function and outputs symbolic expressions as the function summary. (3) A compilation module, which translates the symbolic expressions to C code and applies compilation optimization to the C code to generate deobfuscated code.

3.2 Trace Record and Offline Analysis

When running the obfuscated binary, our system records the dynamic trace, which contains instruction sequences and their operations on registers and memory. We use Pin [9], a binary instrument tool developed by Intel, to record the trace. Pin provides instrumentation interface on instruction level for user to insert callback function before or after execution of instructions, which gives the chance to record all context information. Although Pin allows us to run analysis routine upon execution of program, we prefer to record all information to a file and do offline analysis on the trace file. By decoupling trace recording and the analysis, we can apply multiple rounds of analysis to the trace without running the executable repeatedly. Once we have the trace file, we can reconstruct the control flow graph (CFG) and extract virtual instruction handlers.

CFG Construction. As the code is obfuscated with many indirect jumps, reconstruction the CFG statically is very difficult. For instance, below shows the dispatcher code extracted from VMProtect 2.13 (with most junk code deleted as a matter of convenience for readers).

```
mov al, byte ptr [esi-0x1]
mov ecx, dword ptr [eax*4+0x4058da]
mov dword ptr [esp+0x28], ecx
push dword ptr [esp+0x28]
ret 0x2c
```

The dispatcher fetches the opcode of the instruction pointed to by the *VPC*, which is stored in *esi* register, and then jumps to the corresponding handler according to the opcode, which is done by an indirect jump through *push* and *ret* instruction. Without dynamic execution, it would be difficult to determine the target of such indirect jumps. Static analysis tools such as IDA Pro [10] is unable to generate the CFG for such case.

We instead choose to reconstruct the CFG from the dynamic execution trace. The basic steps of reconstructing the CFG from trace are as follows:

Step 1: Initialization of basic blocks. Initialize each instruction of the trace to a basic block. For the sequence of instructions in the trace such as $I_1 I_2 \ldots$, the instructions are initialized to basic blocks $B_1 B_2 \ldots$, respectively; a directed edge is then added from B_1 to B_2 according the instruction order of trace. We then get a directed graph structure.

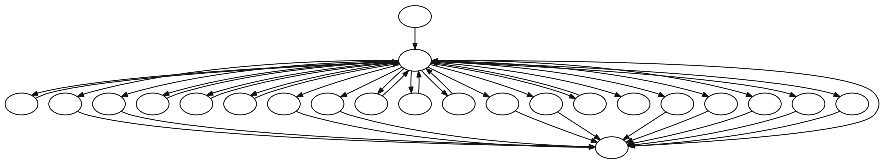

Fig. 2. A typical fetch-decode-dispatch CFG.

Step 2: Emerging basic blocks. For each connected pair of basic blocks B_1 and B_2 are merged into a new block named B_{12}, if and only if the outbound degree of B_1 and inbound degree of B_2 are both 1.

Step 3: Loop. The processing goes back to Step 2 until no more blocks can be merged. Then we get our CFG.

Handler Identification. The CFG of a virtualization-obfuscated binary is characteristic, mainly because of dispatch-based virtual instruction handling. Most obfuscators such as VMProtect and Code Virtualizer, implement the virtual machine as a typical *fetch-decode-dispatch* loop [3], which forms many outgoing branches from the dispatcher node, as shown in Fig. 2. The dispatcher in a virtual machine jumps to a handler function according to the opcode of a virtual instruction and every handler functions return to dispatcher to continue the execution of the next instruction.

According to this feature, we can extract the dispatcher and the handlers by analyzing the CFG. At first, we detect all circle in the graph; each circle in CFG represents a separate execution path of interpretation loop. Secondly, we solve the intersection set of all circles, then the common nodes are marked as dispatcher of interpreter. For each circle, nodes other than dispatcher nodes will be collected sequentially according to execution order and identified as a handler.

3.3 Semantic Analysis of Handlers

Handlers are functions that virtual machine used to interpret virtual instructions. We can extract semantic information of virtual instructions by analyzing corresponding handler functions. We propose to apply a symbolic execution based approach to extracting semantics of each handler.

The overall interpretation logic of virtual machine is too complicated to be symbolically executed as a whole due to time and memory overhead. *Our approach instead applies symbolic execution to handler functions separately.* Each handler function processes a single virtual instruction, which usually has simple logic, therefore the path explosion problem is naturally avoided and symbolic expressions will not be too complex. In addition, with the use of symbol execution, many obfuscations on handler functions such as junk code insertion and instruction replacement are automatically tackled by the symbolic engine.

In our design, all registers and memory in handler function are initialized as symbolic variables. After symbolic execution of the function, the symbolic execution engine outputs a series of symbolic expressions, which represents the operations the handler does.

Here is an example of *vPushReg4* handler in VMProtect. Our prototype system uses Miasm [11] as our symbolic execution engine, which symbolically executes binary code of the handler and returns symbolic expressions in Miasm IR format as below. The disassembly and the symbolic expression are shown below ('@' in Miasm IR expressions means dereference of address):

```
and al, 0x3c
mov edx, dword ptr [edi+eax*1]
sub ebp, 0x4
mov dword ptr [ebp], edx
jmp 0x40100a
```

```
EAX = {(EAX_init[0:8]&0x3C) 0 8, EAX_init[8:32] 8 32}
EDX = @32[(EDI_init + {(EAX_init[0:8]&0x3C) 0 8, EAX_init[8:32]
8 32})]
EBP = (EBP_init+0xFFFFFFFC)
@32[(EBP_init+0xFFFFFFFC)] = @32[(EDI_init+{(EAX_init[0:8]&0x3C)
 0 8, EAX_init[8:32] 8 32})]
```

From the expressions we know that this handler loads the value at a memory address based by *edi* offset by *eax* & 0x3C, then stores the value to where *ebp* points to. Assuming *edi* represents the initial address of the virtual register array and *ebp* the virtual stack top, this handler simply pushes a virtual register value to the virtual stack.

The above example shows that the symbolic expression of the handler function can fully express its operations; that is, it can capture the semantic information of the corresponding virtual instruction.

3.4 Compilation and Code Recovery

While function summary can eliminate dead code *within* a handler, it is ineffective in handling obfuscations *among* virtual instructions, which is one of the main challenges of deobfuscation we aim to conquer. When CISC instructions (e.g., x86) are converted into virtual RISC instructions, the problem is more severe, as a single CISC instruction is usually transformed into multiple virtual RISC instructions. It will significantly benefit code analysis and understanding if we can convert the multiple virtual RISC instructions back into the single CISC one. An intuitive approach is to prepare a whole set of templates for transformation, each of which tries to match a specific sequence of RISC instructions and recover the original CISC instruction. Such template-based transformation has multiple drawbacks: (1) A lot of tedious work has to be done to prepare the transformation templates; worse, whenever a new VM is encountered, they have to be updated. (2) During the transformation, the register information gets lost and more inference must be done to restore it. (3) Obfuscators often apply additional obfuscation on the virtual-instruction layer, which may render the template match ineffective.

We consider such obfuscation as an opposite process of optimization, since it replaces the original concise instructions with more complex but semantically equivalent code, while deobfuscation aims to optimize away the intermediate variables and redundant instructions introduced by virtual machine. Thus, we propose to use modern compiler such as *gcc* and *clang*, which have been developed for years and proven to have excellent optimization capabilities, to optimize the virtual instructions by relying on their built-in data flow analysis and live variable analysis; this way, we can remove redundant instructions and generate concise code.

Translation of Symbolic Expressions. After the previous semantic analysis (Sect. 3.3), we have automatically generated symbolic expressions for each virtual instruction handler. But these symbolic expressions are still unable to be directly processed by the compiler. We have implemented a *Miasm translator* module to translate them to C code, which hence can be processed by compilers. Let us continue the example in Sect. 3.3. The symbolic expressions of *vPushReg4* are translated to C code:

```
*(uint32_t *)(EBP + 0xfffffffc) = *(uint32_t *)(EDI + EAX & 0x3c);
EBP = EBP + 0xfffffffc;
```

The registers are treated as unsigned integer variables, and are converted to pointers when used as addresses.

Compilation Optimization. The following optimizations are conducted to obtain concise and clear code: (1) symbolic variables are translated to local variables if possible; (2) the virtual stack in a virtual machine is converted to a local array variable, and stack pointers now point to array elements; and (3) every virtual instruction handler is defined as an inline function or C macro, and hence the entire bytecode sequence will be translated to a sequence of calls to inline functions.

Let us take VMs in VMProtect as an example: VMProtect reuses the x86 call stack as its virtual stack with *ebp* as stack pointer. When converting to C code, we define a virtual stack as a large enough local array and the stack pointer as a pointer to array elements. Figure 3 is a sample of the converted C code.

In this example, as the handler functions are defined as inline functions, which are invoked by a global function *vmp_func*, the compiler will automatically optimize the generated code. An example on push and pop optimization is shown as Fig. 4. After compilation optimization the redundant stack operation code is eliminated, which makes the result code more concise and easier to understand.

```
#define STACK_SIZE 2048
static uint8_t *sp;

// handlers translated from
// symbolic expressions.
static inline void
vPushReg4(uint32_t *reg_ptr){
  sp -= 4;
  *((uint32_t *)sp) = *reg_ptr;
}
static inline void
vPopReg4(uint32_t *reg_ptr){
  *reg_ptr = *((uint32_t *)sp);
  sp += 4;
}
```

```
void vmp_func()
{
uint32_t regs[16];
uint8_t stack[STACK_SIZE];
sp = &stack[STACK_SIZE/2];

/** virtual instructions **/
...
vPushReg4(&regs[0]);
vPopReg4(&regs[1]);
...
}
```

Fig. 3. A sample of the converted C code.

Before compiling **After inline** **After optimization**

```
void vmp_func()        void vmp_func()                 void vmp_func()
{                      {                               {
...                    ...                             ...
                       sp -= 4;
vPushReg4(&regs[0]);   *((uint32_t *)sp) = regs[0];    regs[1] = regs[0];
vPopReg4(&regs[1]);    regs[1] = *((uint32_t *)sp);    // sp not changed.
                       sp += 4;
...                    ...                             ...
}                      }                               }
```

Fig. 4. An example on push and pop optimization.

4 Evaluation

We have evaluated our system against VMProtect and Code Virtualizer; both are well-known commercial obfuscators. We first performed some micro-benchmark experiments, which consider the following four samples:

- *mov*, which has only a single mov instruction.
- *binop*, which tests binary operators like *add, sub, and, or* and so on.
- *xor*, which is a sample of an xor encoder.
- *base64*, which is a sample of a base64 encoder.

We recorded the count of original and obfuscated instruction trace, dispatcher address, count of handlers and deobfuscated instruction trace. As shown in Table 1, we applied our method to the four samples obfuscated by VMProtect; plus, the base64 sample is obfuscated by different obfuscators as shown in Table 2. According to the results of manual inspection, our method correctly

Table 1. Evaluation against different samples obfuscated by VMProtect.

Sample	Original trace	Obfucscated trace	Dispatcher address	Handlers	Deobfuscated trace
mov	1	357	0x40320b	5	7
binop	128	10276	0x4074f9	20	304
xor	667	123366	0x407592	20	3123
base64	485	107998	0x407063	21	2876

Table 2. Evaluation against base64 sample obfuscated by different obfuscators.

Obfuscator	Original trace	Obfucscated trace	Dispatcher address	Handlers	Deobfuscated trace
VMProtect 1.81	485	107998	0x407063	21	2876
VMProtect 2.13	485	1028805	0x4074f9	22	3445
CodeVirtualizer 1.3.8	485	1007378	0x405556	36	3122

```
// Original source.

void vmp_func(int x[],
int y[], char z)
{
  y[0] = x[1] + x[0];
  y[1] = x[2] ^ x[1];
  y[2] = x[3] & x[2];
  y[3] = x[4] | x[3];
  y[4] = x[4] - x[5];
  y[5] = x[5] << z;
  y[6] = x[6] >> z;
  y[7] = x[7] / x[8];
  y[8] = x[8] % x[9];
  y[9] = x[10] * x[9];
}
```

(a)

```
// Deobfuscated code.
// (decompiled by IDA Pro.)
signed int vmp_func()
{
  int v0; // eax@1
  signed int result; // eax@1
  int *v2; // [sp+440h] [bp-40Ch]@0
  int *v3; // [sp+444h] [bp-408h]@0
  char v4; // [sp+448h] [bp-404h]@0

  *v3 = v2[1] + *v2;
  v3[1] = (v2[2] | v2[1]) & ~(v2[1] & v2[2]);
  v3[2] = v2[3] & v2[2];
  v3[3] = v2[4] | v2[3];
  v3[4] = ~(v2[5] + ~v2[4]);
  v3[5] = v2[5] << v4;
  v3[6] = v2[6] >> v4;
  v3[7] = v2[7] / v2[8];
  v0 = v2[8] % v2[9];
  dword_804A130 = 0;
  v3[8] = v0;
  result = 36;
  v3[9] = v2[10] * v2[9];
  return result;
}
```

(b)

Fig. 5. Source and deobfuscated code of binop.

locates the dispatcher address and extracts the handlers from the trace. We use *gcc* as our compiler with O3 optimization level against the translated virtual instruction trace; the results demonstrate that the trace count is greatly reduced after deobfuscation.

Figure 5 shows an intuitive comparison when we applied our method on the *binop* sample, whose source code is shown Fig. 5(a). After optimizing with compiler and decompiling with IDA Pro, the deobfuscated the pseudo C code, shown as Fig. 5(b), is concise and equivalent to the original code. Basic array memory access and most binary operations such as *add, and, or, shift* etc. between array elements are precisely recovered. The *xor* and *subtraction* operators are not exactly same as origin but the deobfuscated code has equivalent semantic.

5 Related Work

Deobfuscation approaches for virtualization-based obfuscated binaries have long been part of the state-of-the-art research in reverse engineering and binary analysis. Rolles [2] points out the essence of virtualization obfuscation is bytecode interpretation, and his paper describes a generic approach to defeating such protection by completely reversing the emulator, however, no automated system is presented. Coogan et al. [4] present a semantics-based approach to deobfuscating virtualization-obfuscated software. In that work, obfuscated binary is executed and all instructions execution information are recorded in a trace file; then, instructions that interact with the system directly or indirectly are kept with other instructions discarded. It does not perform further deobfuscation, though. Sharif et al. [3] propose an automatic analysis method to extract the virtual program counter information and construct the original control flow graph from a virtualized binary; Kalysch et al. [12] present VMAttack, an IDA Pro plugin, as an assistance tool for analyzing virtualization-packed binaries. Both are new analysis methods but unable to recover the deobfuscated code. While Yadegari et al. [13] pointed out *compiler optimization* can assist deobfuscation, specifically *arithmetic simplification* in their case (Sect. III.C), they did not reuse any compilers as a generic approach to deobfuscation. Instead, they use taint analysis to identify instructions for value propagation, and various specialized optimizations for simplifying the code; plus, symbolic execution is used to generate inputs for running a binary. To our knowledge, our system is the first that reuses modern compilers and leverages compilation optimization as a generic approach to deobfuscating virtualization-obfuscated code.

6 Conclusion

Virtualization obfuscation has been proven to be one of the most effective techniques to obfuscate binaries. This paper presents a novel automated deobfuscation method. It first constructs a CFG via offline trace analysis to detect dispatcher and handler functions, and symbolically execute the handlers to generate symbolic expressions. Then, symbolic expressions are translated into C code and bytecode is converted into invocations of the C functions, which are then optimized by compilers to recover simplified and semantically equivalent code. We have implemented a prototype system and evaluated it against popular commercial obfuscators. The experimental result indicates that our system can successfully recover concise code from virtualization-obfuscated code. Our work demonstrates that compilation optimization is an effective and generic approach to tackling virtualization obfuscation.

Acknowledgments. This work was supported in part by National High Technology Research and Development Program of China (No. 2015AA016004), the National Key R&D Program of China (No. 2016QY04W0802).

References

1. Nagra, J., Collberg, C.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education, London (2009)
2. Rolles, R.: Unpacking virtualization obfuscators. In: 3rd USENIX Workshop on Offensive Technologies (WOOT) (2009)
3. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 94–109. IEEE (2009)
4. Coogan, K., Lu, G., Debray, S.: Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 275–284. ACM (2011)
5. HexEffect: Virtual deobfuscator. http://www.hexeffect.com/virtual_deob.html
6. VMProtect: Vmprotect software protection. http://vmpsoft.com/
7. Oreans: Code virtualizer. https://oreans.com/codevirtualizer.php
8. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: SoK: deep packer inspection: a longitudinal study of the complexity of run-time packers. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 659–673. IEEE (2015)
9. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Not. **40**, 190–200 (2005)
10. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. No Starch Press, San Francisco (2011)
11. CEA: cea-sec/miasm: reverse engineering framework in Python. https://github.com/cea-sec/miasm
12. Kalysch, A., Götzfried, J., Müller, T.: VMAttack: deobfuscating virtualization-based packed binaries. In: ARES (2017)
13. Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A generic approach to automatic deobfuscation of executable code. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 674–691. IEEE (2015)