DynOpVm: VM-based Software Obfuscation with Dynamic Opcode Mapping*

Xiaoyang Cheng[1], Yan Lin[2], Debin Gao[2], and Chunfu Jia[1]

[1] Nankai University, Tianjin, China
chengxiaoyangcxy@outlook.com, cfjia@nankai.edu.cn
[2] Singapore Management University, Singapore
{yanlin.2016, dbgao}@smu.edu.sg

Abstract. VM-based software obfuscation has emerged as an effective technique for program obfuscation. Despite various attempts in improving its effectiveness and security, existing VM-based software obfuscators use potentially multiple but *static* secret mappings between virtual and native opcodes to hide the underlying instructions. In this paper, we present an attack using frequency analysis to effectively recover the secret mapping to compromise the protection, and then propose a novel VM-based obfuscator in which each basic block uses a *dynamic* and control-flow-aware mapping between the virtual and native instructions. We show that our proposed VM-based obfuscator not only renders the frequency analysis attack ineffective, but dictates the execution and program analysis to follow the original control flow of the program, making state-of-the-art backward tainting and slicing ineffective. We implement a prototype of our VM-based obfuscator and show its effectiveness with experiments on SPEC benchmarking and other real-world applications.

Keywords: Frequency analysis, software obfuscation, virtualization

1 Introduction

Unauthorized code analysis and modification threaten the software industry with more sophisticated program analysis and reverse engineering techniques in recent years [5, 8, 35, 36, 38]. Such attacks can lead to undesirable outcomes including unauthorized use of software, cheating in computer games, or bypassing and redirecting payment processes. Program protection and software obfuscation have been key techniques in fighting against such attacks, in which code virtualization using a Virtual Machine (VM) embedded inside an executable is emerging as a promising technique for code obfuscation, e.g., VMProtect[3].

VM-based code obfuscation replaces native instructions in an executable with virtual ones that are uniquely defined by the obfuscator. Such virtual instructions will then be translated into native ones at runtime for correct execution

* This project is partly supported by the National Natural Science Foundation of China (No.61772291) and the Science Foundation of Tianjin (No.17JCZDJC30500).
[3] VMProtect Software protection. http://vmpsoft.com/

with the original semantics. VM-based obfuscation is effective in hiding two aspects of the execution, namely the instructions to be executed (controlled by the secret mapping between virtual and native bytecodes and the handlers) and the execution path (controlled by the dispatcher).

In this paper, we first propose a simple yet effective attack exploiting the static mapping between virtual and native instructions. Our attack is inspired by the frequency analysis of symbols widely employed in crypto-analysis techniques. Our observation is that the native and corresponding virtual instructions would present the same frequency profile with the static mapping, even if the mapping is unknown and well protected by the VM — analogous to the relation between plaintext and ciphertext symbols whose mapping could be unknown but their frequency profiles are identical. We show that our frequency attack enables an attacker to recover the mapping between virtual and native instructions, which compromises the handlers in the VM embedded. Note that although more recent and enhanced VM-based protectors use multiple mappings between virtual and native instructions, the statically defined (secret and multiple) mappings only add more complexity to the frequency analysis but do not render it ineffective.

Keeping this effective attack in mind, we propose a novel VM-based software protection called DynOpVm, in which the mapping between virtual and native instructions is *dynamic* and *control-flow aware*. The dynamic nature of the mapping renders frequency attack ineffective since every protected basic block employs a different mapping between the virtual and native instructions. The control-flow-aware protection ensures that the correct mapping can only be recovered by following the correct control flow execution, which dictates the execution and, more importantly, the analysis of the program, to follow the original control flow. This further makes program analysis, in particular, backward tainting [2,9,13,20] and slicing techniques [43], difficult as the instructions cannot be decoded at the middle of any program execution.

We face a number of technical challenges especially in designing the control-flow-aware mapping between virtual and native instructions. One of them is to support basic blocks with *multiple* control flows which could result in *multiple* mappings — a conflict since each basic block can only be encoded using a single mapping. We propose solving this challenge by utilizing the secret sharing algorithm, enabling a *single* mapping to be derived from *multiple* control flows. We also demonstrate the effectiveness of our frequency attack and DynOpVm with experiments with the SPEC benchmarking and other real-world applications.

2 Background and Related Work

2.1 VM-based program protection

Here we take the example of Rewolf virtualizer[4] (due to its available source code for clear understanding and experimentation) and briefly describe how an executable protected by it works (with its add-on layer `Poly` disabled). As shown

[4] X86 virtualizer. http://rewolf.pl/

in Fig. 1, when program execution comes to any protected code, a control transfer directs execution of the program to a dispatcher, which obtains the potential virtual instruction and checks its prefix. All virtual instructions begin with a unique prefix (`0xFFFF` in the case of Rewolf virtualizer) as an indicator to the VM. After confirming the identity of the virtual instruction, the VM invokes a corresponding handler (dictated by the virtual opcode that is the next byte in the virtual instruction) to perform the corresponding operation of the original native instruction.

Following the idea of code virtualization, a number of VM-based code obfuscation approaches have been proposed. These include methods in securing the VM [3, 14, 46] and improving the obfuscation process [15, 42]. Publicly available tools like VM-Protect, Code Virtualizer[5] and Themida[6] also employ special protections for runtime environments, e.g., VOT4CS [4].

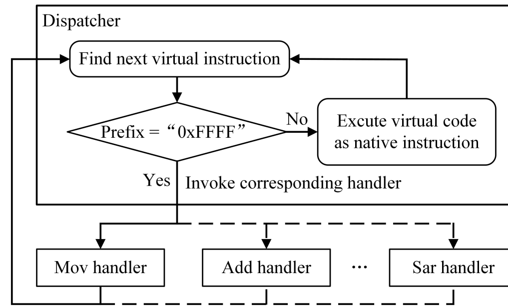


Fig. 1: VM in existing obfuscators

Many of these existing VM-based program obfuscators use a single mapping between virtual and native instructions. Kuang et al. [22] used different ways to interpret the same virtual instructions and obfuscated the atomic handlers. Although the handlers are obfuscated, there still remains only one mapping between native and virtual instructions. Other VM-based obfuscators, e.g., VMProtect, maintain multiple mappings between virtual and native instructions, and randomly choose one of them in each obfuscation instance; however, the multiple available mappings were statically designed with limited variations.

2.2 Attacks on VM-protected programs

Rolles proposed to reverse engineer the VM in order to convert virtual bytecode to native instructions [31]. Based on this idea, Rotalumé [34] was further proposed to detect the mapping between virtual bytecode and handlers. Guillot et al. [16] automatically search for patterns of obfuscation. Similarly, VMAttack [19] was presented as an automatic deobfuscation tool to analyze VM structure and to compress instruction sequences. Coogan et al. [11] applied equational reasoning [10] to reconvert the native code. Other proposals [40,45] utilized taint analysis to reveal the dependency of virtual code and the embedded VM. BinSim [26] attacked the code virtualizer with the help of backward slicing. VMHunt [44]

[5] Code virtualizer. https://oreans.com/codevirtualizer.php
[6] Themida. https://www.oreans.com/themida.php

tackled the problem using tracing, symbolic execution, and backward slicing. Our proposed frequency attack works on a different dimension in that it avoids analyzing the semantics of the VM or its corresponding handlers.

2.3 Instruction-Set Randomization and Control-Flow Carrying Code

Instruction-Set Randomization (ISR) can also be seen as a VM-based system, and was proposed as a mitigation against code-injection attacks [6,21,27]. It uses an execution environment to interpret and execute a randomized instruction set which is unique for each program. There has also been proposals to use ISR to enforce CFI [12, 37, 39]. Instead of a unique instruction set for each program, DynOpVm makes use of a unique instruction set for each basic block by generating a unique secret from each control transfer. Similar to ISR, Control-flow Carrying Code (C^3) [23] uses a dynamic instrumentation system to assist CFI-enforced execution of a program. DynOpVm shares the same idea with C^3 on using secret sharing to encode/encrypt binary instructions; however, DynOpVm and C^3 are based on different threat models, are proposed to fight against different types of attacks, and are implemented in completely different ways. DynOpVm fights against frequency analysis on VM-based obfuscators, while C^3 is to counter Data-Oriented Programming attacks on traditional CFI systems. DynOpVm produces self-contained VM-embedded executables which can execute directly on mainstream Linux systems, whereas C^3 requires a dynamic instrumentation systems for its execution.

3 Frequency attacks on VM-based Program Obfuscation

As discussed in Section 2.1, existing VM-based program obfuscators, including the original work and subsequent enhancements [3, 14, 15, 22, 31, 42, 46], use a secret but static mapping (potentially multiple ones) between virtual and native instructions for obfuscation. Intuitively, such static mappings, although secret and unknown to an attacker, make frequency analysis possible since native instructions present some unique and specific frequency profile in normal programs.

3.1 Frequency profile of native instructions

A prerequisite of our attack is a unique frequency profile exhibited by native instructions. Related frequency analysis has been conducted on different platforms since the last century [1, 17, 18, 29, 30, 32], most of which focus on runtime statistics of the instruction set. On the other hand, the objective of our frequency analysis is to collect static profile of instructions for program analysis.

We statically analyze the number of occurrences of native instructions in executables under directory `/bin` on 64-bit Ubuntu 18.04 and present the 15 instructions with the highest frequencies in Fig. 2. We notice that this frequency profile is uneven while consistent with low standard deviation among the 128

executables. For example, `mov` shows up most often with its frequency more than 3 times of the second most frequent instruction `call`. In addition, `mov` presents a frequency of over 30% while many other instructions have frequencies of lower than 1%. Although these other instructions with low frequency do not stand out in the profile, we comment that the decoding process usually requires only a few instructions to be identified as bootstraps, and other instructions could then be easily recovered by, e.g., frequency analysis of instruction subsequences.

3.2 Frequency analysis as an attack

We first implement a virtualizer compatible with 64-bit Linux executables using the same strategy as Rewolf Virtualizer[7], and apply it to protect selected code pieces in SPEC CPU2006 benchmarking programs. We note that other variations of the obfuscator may differ in the implementation details, e.g., the commercial product VMProtect that is closed-source, but we have not noticed evidences that such differences render our frequency attack ineffective, including the fact that it randomly chooses from multiple static handlers.

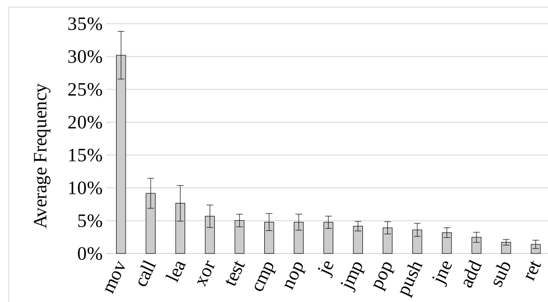
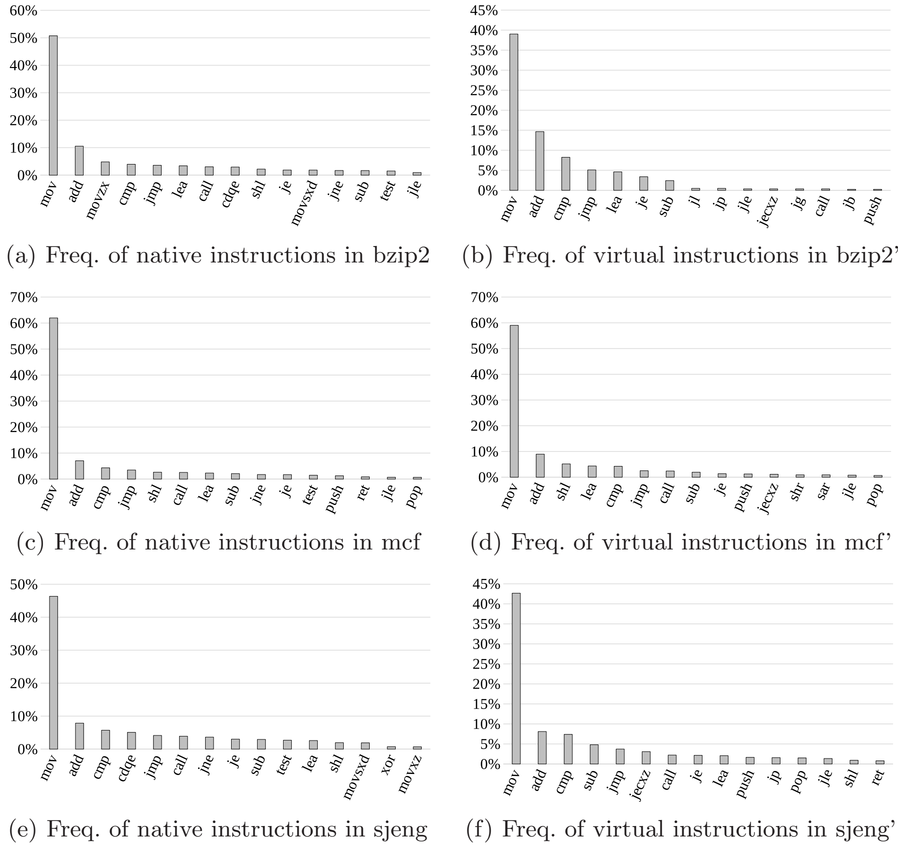


Fig. 2: Freq. analysis of 128 binaries

We intentionally select small code pieces for protection to see if that renders the frequency analysis less accurate. We search for the prefix of `0xFFFF` to identify all virtual instructions and their virtual opcode (the byte following the prefix). Note that other obfuscators may employ more complicated ways of encoding the virtual opcodes; however, existing work (e.g., VMHunt [44]) has shown that the beginning of various handlers could be effectively located with analysis of context switch patterns, which clears another prerequisite of our frequency analysis. Here we only present results for 3 programs, `bzip2` (11182 bytes of code protected), `mcf` (3058 bytes of code protected), and `sjeng` (5510 bytes of code protected).

Fig. 3 shows the analysis result of 20 instructions with the highest frequency for both the original program and that protected by our virtualizer. It also shows the ground truth mapping between corresponding virtual and native instructions. Results show that frequency analysis attack is accurate even for small code pieces. For example, the top two instructions are always mapped correctly, while the overlapping of top 10 instructions between the original and the protected programs cover 8 instructions or more.

[7] We are not aware of any 64-bit VM-based obfuscator that is open source, and therefore decide to make one based on the 32-bit Rewolf virtualizer.



(a) Freq. of native instructions in bzip2 (b) Freq. of virtual instructions in bzip2'

(c) Freq. of native instructions in mcf (d) Freq. of virtual instructions in mcf'

(e) Freq. of native instructions in sjeng (f) Freq. of virtual instructions in sjeng'

Fig. 3: Frequency analysis on programs protected by Rewolf virtualizer

When considering a slightly different threat model in which the attacker does not have any information of the protected program (maybe in the event that the entire program is protected) and therefore can only compare the frequency analysis result (Fig. 3(b), Fig. 3(d), and Fig. 3(f)) with the general statistics of native instructions (Fig. 2), we obtain similar results — we can identify the most frequent instruction `mov` with at least 6 overlappings in the top 10 instructions.

While there are other ways of improving this attack, e.g., by analyzing operands of the instructions and other context information, we believe that the simple demonstration above is sufficient to reveal this fundamental weakness of the existing VM-based program obfuscators, in which static mappings (although secret) are used between virtual and native instructions. To improve the security and to fight against such frequency attacks, we propose a novel technique that employs dynamic and control-flow-aware mappings; see Section 4.

3.3 Threat model and assumptions

Our objective is to propose a novel VM-based program obfuscator that renders the frequency analysis attack and state-of-the-art program analysis methods, e.g., backward tainting and slicing techniques, ineffective. We assume that the attacker is aware of the details of our technique and has access to the protected binary executable. The attack can be Man-At-The-End (MATE) attack and leverage memory disclosure vulnerabilities in the target application to read and analyze the memory, including data and the code section of the target program.

4 Design and Implementation of DynOpVm

Section 3 presents a simple yet effective attack on existing VM-based program obfuscators using frequency analysis. In this section, we present our novel VM-based obfuscator that is resistant against such attacks. Moreover, to defend against other attacks as discussed in Section 2.2, we have a second objective of rendering program analysis techniques, in particular, backward tainting and slicing techniques, ineffective on the protected program (piece).

4.1 Overview of DynOpVm

Defending against frequency analysis is a well-explored problem in cryptography. For example, Vigenère cipher was proposed as a poly-alphabetic substitution system to fight against frequency analysis on English letters [7,24], with the idea that the same plaintext letters can be encrypted to different ciphertext letters. Our proposed solution is inspired by this simple idea to construct different and dynamic mappings between virtual and native instructions for different basic blocks, even if various basic blocks contain the same native instruction.

Fighting against state-of-the-art program analysis tools like backward tainting and slicing is more challenging. What makes such backward analysis possible is the "two-way" nature of control-flow information presented in normal executables, i.e., it is easy to find both the predecessor and successor of an instruction. Essentially we want to make control-flow information in the protected program "one-way", in that even if an attacker manages to decode specific virtual instructions, we want to make it difficult to reveal the caller[8] basic block. We leave it as future work to make even the forward analysis difficult, since the program needs to be able to execute in a forward manner absent from analysis.

Our solution is to make the mapping between virtual and native instructions dependent on control flows, i.e., the addresses of caller and callee instructions. Since both addresses are available in a forward execution (e.g., in executing `call $0x400460` in Fig. 4, the caller and callee addresses are `0x400450` and `0x400460`), reconstructing the mapping between virtual and native instructions and decoding the callee is easy. On the other hand, backward analysis to figure

[8] In the rest of this paper, we use the words "caller" and "callee" to refer to predecessor and successor basic blocks in a control transfer.

out the caller of `call $0x400460` is difficult since it uses a mapping that is determined by its own caller (address of `jmp $0x400450`).

We design and implement a prototype of our novel VM-based program obfuscator called DynOpVm following this idea. DynOpVm takes as input the original binary executable (without source code), statically encodes each basic block into virtual instructions with a mapping uniquely determined by the caller and callee addresses in the control transfer, and inserts a VM to decode basic blocks. Control transfers are redirected to the VM which dynamically reconstructs the specific mapping between virtual and native instructions, decodes the next basic block "on-the-fly", and then continues with the valid control transfer. We present our detailed design and implementation in the next subsections.

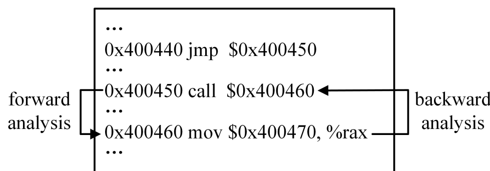
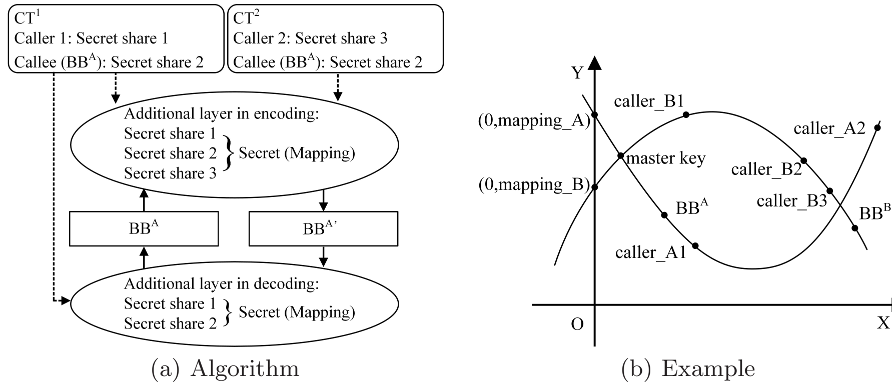


Fig. 4: Forward and backward analyses

4.2 Control-flow-aware encoding of basic blocks

As discussed in Section 4.1, DynOpVm statically performs binary rewriting. To stay focused in this paper, we make use of existing tools for static analysis and rewriting, and consider general challenges (e.g., distinguishing code from data) out of our scope. At a first glance, such a process isn't overly complicated; however, a basic block could have *multiple* callers, which will result in *multiple* mappings between virtual and native instructions derived for the same callee block. On the other hand, each callee block could only be encoded with one unique mapping. The challenge here is to derive the same mapping from multiple control transfers with multiple callers. Our solution is to introduce an additional layer in deriving the mapping, where each source or destination address of valid control transfer determines a secret share, and multiple secret shares could be used to reconstruct the same mapping — a typical application of Shamir's secret sharing algorithm [33]. In Fig. 5(a), the callers of control transfers CT^1 and CT^2, both of which target BB^A, contribute two different secret shares. Both are used to compute the same secret together with the secret share generated from callee address (the address of BB^A). The secret is then used to encode BB^A. Note that at runtime, only one of the secret shares from the two callers is used to derive the mapping, which is well supported by the secret sharing algorithm.

In applying secret sharing, caller and callee addresses constitute two points on a secret sharing polynomial. We introduce a third point as a master key randomly chosen to defend against information disclosure attacks which could potentially be exploited to reconstruct the mapping. DynOpVm takes a configuration with $t = 3$ (a parabola) to enable reconstruction of the mapping with (potentially multiple) valid control transfers. Fig. 5(b) shows two parabolas: one



(a) Algorithm (b) Example

Fig. 5: Secret sharing

representing a basic block BB^A with two valid callers, and the other representing BB^B with three valid callers. The intersection of the parabola with the y-axis is the secret to determine the mapping between virtual and native instructions for the corresponding callee. DynOpVm obtains the X and Y coordinates (k bits) of a point from the lower-order odd- and even-index bits of an address. The master key (of $2k$ bits long) is randomly chosen. We discuss the security and performance implication of the choice of value k in Section 5.

Although this algorithm well supports multiple callers, it introduces constraints on the addresses. For example, once the master key, the address of the callee, and that of one caller are determined, the parabola is fully established and addresses of the remaining callers have to be on the curve. This results in constraints in our binary rewriting to redistribute the basic blocks:

- "Call-preceded" basic blocks (those followed by `call`) cannot be redistributed freely as they are the targets of `ret` instructions. Such additional constraints could result in an unsolvable layout of basic blocks. Our solution is to replace all `call` instructions with `push` followed by `jmp` to remove such additional constraints. A similar challenge arises for conditional jumps and their fall-through instructions, which can be resolved with the same idea.
- Parabolas can have at most two intersections, one of which is the master key. This means that two different callees may only have up to one common caller — an invalid assumption in many applications. To handle this, we add intermediate "stub" blocks to remove the additional common callers.
- Basic blocks with multiple entries would result in multiple mappings derived. We make copies of them to ensure that each basic block has only one entry.

To redistribute basic blocks, we use a look-ahead depth first search algorithm to avoid circular constraints, e.g., when two callers of a to-be-redistributed basic block with fixed addresses make it impossible to find a valid parabola.

DynOpVm encodes and decodes between virtual and native instructions with a simple XOR operation with the secret derived from the secret sharing algo-

rithm. This design of the mapping between virtual and native instructions is mainly due to its simplicity and efficiency. After every basic block is redistributed and encoded with the corresponding mapping, we can then embed the VM and insert control transfers to it.

4.3 Embedding a VM

Before making a control transfer to the VM, DynOpVm saves the `rflag` state and uses registers to pass the necessary information to the VM. Such information includes the address of the caller and callee (two possible callee addresses in case of conditional jumps, out of which the VM chooses one depending on the `rflag` value) and the type of control transfer instruction.

The main task of the VM is to reconstruct the secret and to decode and execute the corresponding basic blocks. Our design of the VM consists of three components — a dispatcher, a decoder, and an actuator — which is slightly different from existing techniques of VM-based program obfuscation as discussed in Section 2.1 [3, 6, 15, 25, 46]. Our VM dispatcher makes use of information passed to the VM to obtain the address of the callee. After that, the decoder reconstructs the parabola, computes the secret for the callee, and then decodes the instructions into a buffer dynamically allocated, whose address is stored in a segment register. In the end, the VM actuator transfers control to the decoded (native) instructions and executes them. Fig. 6 shows this process. Note that the VM has two potential control flows from the dispatcher — `d-f-g` and `b-c` — for control transfers to protected and unprotected code, respectively. We discuss more details of our support of this in Section 4.4.

The unknown length of the callee basic block makes it tricky for it to be decoded. DynOpVm uses an optimization to decode a fixed size of 128 bytes at a time and repeats the decoding routine in cases of larger basic blocks. `nop` instructions are inserted for alignment purposes for efficient execution. Another challenge is the conflict with original program code if our code added and VM execution use the stack. DynOpVm uses the `fs` register instead to avoid this conflict.

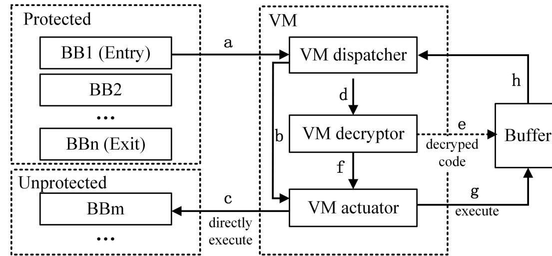


Fig. 6: VM dispatcher, decoder, and actuator

4.4 Supporting partial protection

The key challenge in supporting partial protection of an executable is to make control transfers between protected and unprotected code. This can be achieved by adding control transfers to the VM only in protected basic blocks. However,

such a simple solution may potentially allow dedicated attackers to reconstruct the parabola for a protected entry block that has multiple unprotected callers, since these callers are all on the parabola curve. Combining multiple instances of such attacks could even allow recovery of the secret master key.

We introduce a more secure way to support partial protection to fight against such attacks. The basic idea is to reduce the number of unprotected callers with code cloning and inlining. DynOpVm makes a copy of the chosen unprotected code to be inlined into the protected region to reduce the number of control transfers between protected and unprotected code. In this way, attackers will find fewer points on the parabola curve to reconstruct the secret. DynOpVm also maintains a list of valid exit targets in the VM to allow/disallow transfers to unprotected code at runtime. We further propose two potential solutions that could avoid the need of cloning and inlining, since inlining may not be a practical solution in interfacing with, e.g., system libraries. Assuming that the protected code P_{vt} transfers control to a system library function lib with a basic block $\mathsf{BB}^{\mathsf{call}}$, and control returns to P_{vt} at basic block $\mathsf{BB}^{\mathsf{ret}}$, the two solutions are:

1. Leaving $\mathsf{BB}^{\mathsf{ret}}$ and lib unprotected (with $\mathsf{BB}^{\mathsf{call}}$ protected) while adding the address of $\mathsf{BB}^{\mathsf{ret}}$ as a valid exit target maintained by the VM.
2. Leaving lib unprotected while having $\mathsf{BB}^{\mathsf{ret}}$ protected with a parabola that passes though the origin, which means that $\mathsf{BB}^{\mathsf{ret}}$ is encoded with key 0.

Both solutions allow proper execution of the program with more basic blocks exposed in plaintext, although it is non-trivial for an attacker to differentiate them from those encoded with nonzero keys. Solution (1) allows $\mathsf{BB}^{\mathsf{ret}}$ to be the target of control transfers from any protected basic block. Solution (2) restricts control transfers to $\mathsf{BB}^{\mathsf{ret}}$, but potentially allows an attacker with the capability of launching memory disclosure attacks to recover the master secret by reconstructing the parabola, since an additional point (the origin) and the plaintext instruction inside $\mathsf{BB}^{\mathsf{ret}}$ is given to the attacker.

DynOpVm assumes a strong threat model where memory disclosure attacks are assumed possible, and therefore uses the solution of code cloning and inlining for better security. We comment that the above two solutions could be useful under a different threat model.

4.5 Implementation

We implemented a prototype of DynOpVm for Linux x64 platform. DynOpVm takes as input a 64-bit ELF binary and outputs a modified binary executable with selected basic blocks encoded into virtual instructions and VM embedded. The static instrumentation component is implemented as 8,200 lines of python code with the help of Capstone [28] for disassembling and Type-armor [41] for constructing the CFG. The VM interpretation and execution component is implemented as 900 lines of assembly instructions inserted into the executable file.

Besides executing the design presented in earlier subsections, DynOpVm makes use of gaps among redistributed protected basic blocks to host unprotected functions, and fills the remaining gaps with `nop` instructions. Finally,

DynOpVm patches the new binary file with the text segment extended and corresponding addresses (code pointers, function pointers, data pointers, jump tables and virtual tables) and section information updated.

One challenge is to deal with instructions with PC-relative addressing since the execution will be in the buffer dynamically allocated and the program counter (%rip) at runtime is unknown at static instrumentation. Our solution is to remove PC-relative addressing mode during binary rewriting. To support multi-threaded programs, we use a new memory page for decoding basic blocks for each thread by checking the value in `fs:0x158` where we store the buffer address.

5 Evaluation

In this section, we evaluate the security of DynOpVm with regards to frequency analysis and Shannon entropy, and apply backward slicing attacks presented by Ming et al. [26, 44] to evaluate its resistance to such analysis. Besides that, we measure the performance overhead of DynOpVm with real-world applications.

5.1 Security evaluation

Frequency attack As shown in Section 3, existing VM-based program obfuscators like Rewolf virtualizer suffer from frequency analysis which allows an attacker to easily figure out the mapping between virtual and native instructions. Intuitively, DynOpVm encodes each basic block with a different mapping determined by the control transfer, and is resistant to such attacks.

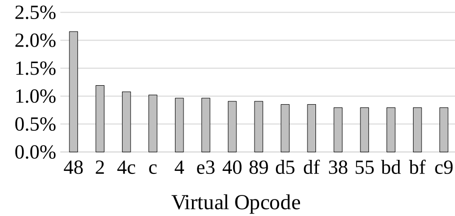
Moreover, the use of XOR operation in encoding instructions effectively removes any obvious patterns as in some existing VM-based program obfuscators, e.g., `0xFFFF` in the Rewolf virtualizer. Lack of the capability of identifying each virtual instructions, attackers could not even perform the frequency analysis on them. Here, we want to see how far the frequency analysis could go even if attackers could identify the start of every virtual instruction, and present the results of such frequency analysis[9] on the same SPEC benchmarking programs as used in Section3; see Fig. 7.

Comparing graphs in Fig. 7 and those in Fig. 3 reveals two observations. First, the shape is very different in the sense that the frequency values decay a lot faster in unprotected programs and those protected by the Rowolf virtualizer, while they decay a lot more slowly in programs protected by DynOpVm. Second, the peak frequency value for unprotected programs and those protected by Rowolf virtualizer is at 40% or more, while that for programs protected by DynOpVm is at most one tenth at 4%. This suggests that programs protected by DynOpVm present a much more even distribution in frequency analysis with many virtual instructions at a non-negligible frequency, making recovering the mapping between virtual and native instructions difficult.

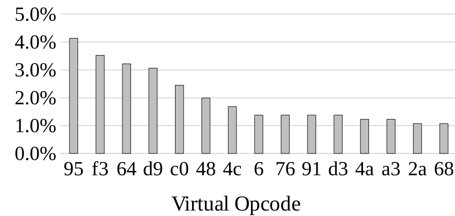
[9] Our frequency analysis here is on the first byte of the virtual instructions, since the length of them is unknown to attackers.

Entropy and randomness analysis To gain an even more intuitive understanding and to consider the entire virtual instruction (as opposed to just the opcode in the frequency analysis), we calculate the Shannon entropy of the SPEC CPU2006 benchmarking programs unprotected, protected by Rewolf, and protected by DynOpVm, which is shown in Fig. 8. Shannon entropy estimates the randomness in the binary information streams — the higher the entropy, the more random the byte stream is.

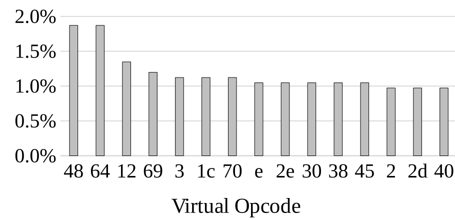
Fig. 8 shows that programs protected by DynOpVm have more random byte streams and therefore are harder to analyze in terms of the frequency distribution or differentiation among virtual instructions. Interestingly, Rewolf virtualizer produces less random byte streams than the unprotected programs, likely due to the prefix `0xFFFF` inserted for every virtual instruction. Note that in this experiment, DynOpVm uses the smallest secret size that makes redistribution of basic blocks possible, with $k \in [8, 10]$.



(a) Freq. of virtual opcodes in bzip2”



(b) Freq. of virtual opcodes in mcf”



(c) Freq. of virtual opcodes in sjeng”

Fig. 7: Frequency analysis of DynOpVm virtual code

Backward tainting/slicing analysis Due to the relatively strong threat model used (Section 3.3), we admit that it is not impossible for an attacker to decode a specific basic block without dynamically executing it. However, an effective attack would require that sufficient information about predecessors of the basic block be known, e.g., addresses of the control transfer instructions of *multiple* predecessor blocks. It could be possible to obtain such information for one of the predecessor blocks, if, e.g., the predecessor block has been successfully decoded or if it is in an unprotected component; however, obtaining such information for multiple predecessor blocks would require that many protected blocks have been previously decoded successfully. Not arming with information of predecessor blocks, an attacker would face the decoding task of the basic block that has been XOR'ed with a key of size k, where $k \in [8, 10]$ in our experiments.

We stress that our objective in DynOpVm is to make program analysis starting from the middle of an execution difficult, e.g., in backward tainting and slicing analysis, where information of the predecessor blocks is typically unavailable. Here we perform an experiment to simulate backward tainting and slicing analysis on a program protected by DynOpVm. We assume that the analysis starts from a basic block BB^0 fully decoded with K^0 (e.g., one that contains an interesting sink instruction). We also assume that it is a strong attacker who had previously obtained the master key used in protecting this binary (probably via some memory disclosure attack). The objective of the attack is to find the predecessor of BB^0, denoted as BB^{-1}, and to have it decoded (find K^{-1}) to reveal its native instructions. Intuitively, the steps involved are as follows.

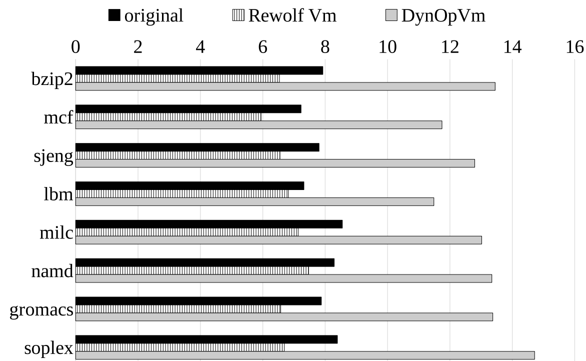


Fig. 8: Shannon entropy

1. Reconstruct the parabola for BB^0 with three points on it: the master key, K^0, the address of the entry point of BB^0.
2. For every point on the parabola constructed (a total of $2^k - 2$ points), derive the corresponding address which is potentially the address of the control transfer instruction of BB^{-1}.
3. For every potential control transfer instruction of BB^{-1}, and for every possible block size of BB^{-1} (we tried all numbers in $[50, 150]$), reconstruct K^{-1} by XORing the virtual and native instructions (assuming that DynOpVm uses a single dedicated native instruction for control transfers).
4. Use the derived K^{-1} to decode the other instructions in BB^{-1} and see if they are valid native instructions.

We follow this strategy to analyze the protected program `bzip2` with a desktop computer with i7-6700 CPU running at 3.40GHz and 16GB of RAM running Ubuntu 64-bit kernel 4.4. Results show that even if DynOpVm uses a dedicated `jmp` instruction for all control transfers, the total number of valid BB^{-1} found per BB^0 on average is 437.03; while, in fact, basic blocks in `bzip2` have on average 1.32 predecessor basic blocks. Moreover, the time it takes to try all possible callers of a single BB^0 is 996.51 seconds on average.

Results show that such an attack is imprecise and inefficient in decoding the predecessor blocks. We note that the experiment above was performed on `bzip2` protected with DynOpVm on a setting of $k = 8$. When k is 9 or 10, the average

numbers of valid predecessor blocks jump to 2,362.79 and 7,258.40, respectively, and the average time it takes to try all predecessor blocks of a single BB^0 is 4,415.07 and 18,268.28 seconds, respectively.

5.2 Performance evaluation

In the performance evaluation, we expand the target set of programs from SPEC benchmarking programs to include an image processing tool `convert` from `ImageMagicks`, two web servers `httpd` and `lighttpd`, a distributed memory caching system `Memcached`, and an FTP server `Pure-FTPd`. We randomly select a few functions in these programs and apply DynOpVm to protect them. Experiments are performed on a desktop computer with Intel Core2 Duo CPU at 3.16GHz and 8GB of RAM running Ubuntu 64-bit kernel 4.4.

For the SPEC benchmarking programs and `convert`, we execute them with standard input data `train` and test cases bundled with the source code, respectively. To benchmark the web servers, we configure Apache Benchmark[10] to issue 2,000 requests with 100 concurrent connections. To benchmark `Memcached`, we use `memslap` benchmark[11] with its default configuration. For the FTP server, we configure `pyftpbench` benchmark[12] to open 20 connections and request 100 files per connection with over 100MB of data requested. We run each experiment 10 times, ensure that the CPUs are fully loaded throughout the tests, and report the median. Table 1 shows the details of these programs where data is collected dynamically at run time. Note that we intentionally have a program (`bzip2`) with more than 99% of the (runtime) instructions protected and other programs with less than 0.1% instructions protected.

Table 1: Details of programs in our performance evaluation set

	# of instructions	# of instructions protected by DynOpVm	Percentage of code protected	# of context switches from unprotected to protected code	# of branching instructions protected by DynOpVm
`bzip2`	311,200,698	310,963,079	99.92%	12,563	11,632,308
`mcf`	6,822,420,892	193,631,839	2.84%	118,645	22,755,914
`sjeng`	27,751,560,742	287,456,204	1.04%	3,169,096	14,012,972
`convert`	18,875,392	16,967	0.09%	187	2,372
`httpd`	292,300,296	864,936	0.30%	3,958	80,843
`lighttpd`	135,150,518	1,825,473	1.35%	16,377	116,225
`memcached`	1,806,983,275	14,789,569	0.82%	456,924	1,341,456
`Pure-ftpd`	710,390,558	505,940	0.07%	2,133	27,262

[10] Apache benchmark. http://httpd.apache.org/docs/2.0/programs/ab.html

[11] Memslap: load testing and benchmarking a server. http://docs.libmemcached.org/bin/memslap.html

[12] Extremely fast and scalable Python FTP server library. https://github.com/giampaolo/pyftpdlib

Since Rewolf virtualizer has limitations in supporting multi-threaded execution, we compare performance overhead of DynOpVm with another VM-based program obfuscator VMProtect[13].

To evaluate the overhead in execution time, we use the default key size $k = 8$ with exception in `httpd` as the `ap_getparents` function has a complicated CFG and requires a key size of 9 for its protection. We expect two main contributing factors to the performance overhead. First is for the VM to allocate memory and to free up memory resources. Second is the reconstruction of the secret and decoding of the target basic blocks. To gain a detailed understanding of the overhead introduced by either factor, we report the execution time of the original programs unprotected, programs protected by DynOpVm with encoding/decoding disabled and enabled, and programs protected by VMProtect with and without packing; see Fig. 9.

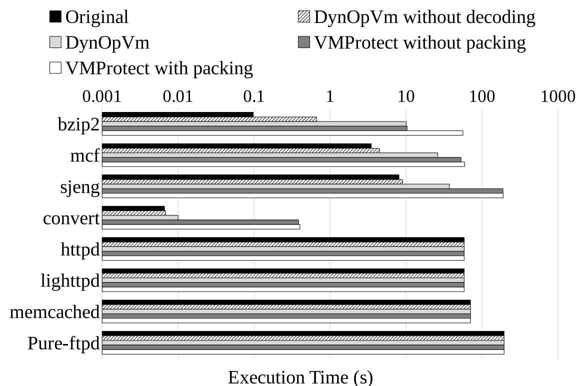


Fig. 9: Overhead in execution time

Our first observation is that programs with more protected code (i.e., the SPEC benchmarking programs) incur higher overhead, which is as expected. Although such overhead could go up to 10 times for DynOpVm when almost 100% of the code is protected, the overhead is negligible when only specific and small amount of code needs to be protected. We comment that this makes DynOpVm practically usable in real-world scenarios. Recent studies [44] also report that most existing VM-based obfuscators target the protection of a small portion of the code only.

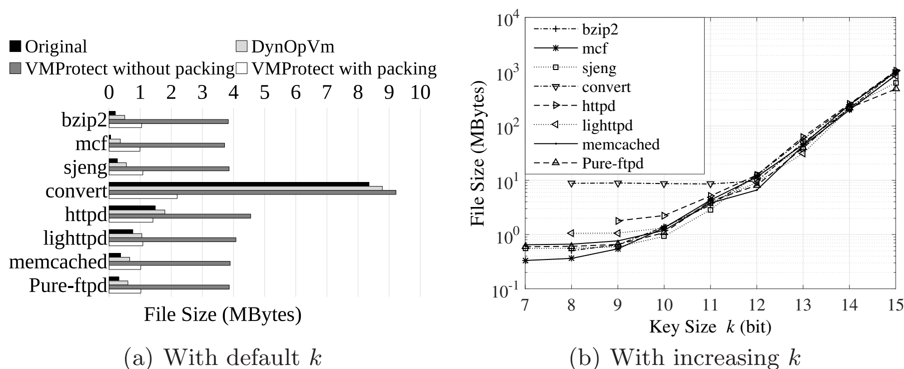
We also notice that the runtime overhead of DynOpVm mainly comes from the decoding of basic blocks, as evidenced by the substantial difference between DynOpVm with and without decoding for the three SPEC benchmarking programs. This is also not surprising as decoding involves reconstructing the parabola and performing the XOR operation, which are heavy in computation.

Our third observation is that the overhead of DynOpVm is noticeably lower than that of VMProtect especially when more code needs to be protected, and this is true even with packing disabled on VMProtect, which makes a fair comparison since DynOpVm does not support packing in its current prototype.

[13] We could not use VMProtect in our frequency analysis and security evaluation due to its close-source nature.

Overhead in file size Since DynOpVm needs to redistribute basic blocks according to the secret sharing function, it may incur considerable overhead in terms of the file sizes. Moreover, this overhead in space may vary according to the different settings of k. For example, when $k = 12$, the address of an instruction can be any value in the range of $(0, 2^{24})$ as both x and y are 12 bits long.

Fig. 10(a) shows the file sizes of the original programs, programs protected by DynOpVm (with $k = 9$ for `httpd` and $k = 8$ for all other programs), and the programs protected by VMProtect (with and without packing enabled). We see that this default setting of k results in DynOpVm having significantly smaller overhead in file size compared to VMProtect when packing is disabled. We stress that the packing option is also potentially possible for DynOpVm, although it is not implemented in our current prototype.



(a) With default k

(b) With increasing k

Fig. 10: Overhead of file sizes

When k increases, DynOpVm gains better protection due to the bigger space in possible mappings between virtual and native instructions. However, it also results in higher overhead in file sizes; see Fig. 10(b). A closer inspection shows that it increases exponentially rather than linearly with increase in k. This demonstrates the trade off in configuring k, and it may favor smaller values in the range of $[8, 12]$ to avoid excessive disk and memory usage.

6 Limitations and Conclusion

Besides the limitation of code cloning and inlining to support partial program protection (note our alternative designs discussed in Section 4.4), the current prototype of DynOpVm stores the master key within the protected executable for its simplicity of implementation. This can be improved with a networked component embedded to retrieve the master key during program execution. Our current prototype also reconstructs the mapping for every basic block at runtime, which could be improved with a cache mechanism.

In this paper, we first present a simple yet effective attack using frequency analysis to recover the mapping between virtual and native instructions, and then design and implement a novel VM-based program obfuscation technique called DynOpVm which employs dynamic mapping between virtual and native instructions that is determined by individual control transfers. DynOpVm is resistant to not only the frequency analysis attack but also state-of-the-art backward taint and slicing program analysis techniques. Our evaluation with real-world applications shows that DynOpVm renders frequency attacks ineffective.

References

1. Adams, T.L., Zimmerman, R.E.: An analysis of 8086 instruction set usage in ms dos programs. ACM SIGARCH Computer Architecture News **17**(2), 152–160 (1989)
2. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices **49**(6), 259–269 (2014)
3. Averbuch, A., Kiperberg, M., Zaidenberg, N.J.: An efficient vm-based software protection. In: Proceedings of the 5th International Conference on Network and System Security (NSS). pp. 121–128. IEEE (2011)
4. Banescu, S., Lucaci, C., Krämer, B., Pretschner, A.: Vot4cs: A virtualization obfuscation tool for c. In: Proceedings of the 2016 ACM Workshop on Software PROtection. pp. 39–49. ACM (2016)
5. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: Byteweight: Learning to recognize functions in binary code. In: Proceedings of the 23rd USENIX Security Symposium. pp. 845–860 (2014)
6. Barrantes, E.G., Ackley, D.H., Palmer, T.S., Stefanovic, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks. In: Proceedings of the 10th ACM conference on Computer and communications security. pp. 281–289. ACM (2003)
7. Bruen, A.A., Forcinito, M.A.: Cryptography, information theory, and error-correction: a handbook for the 21st century, vol. 68. John Wiley & Sons (2011)
8. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: A binary analysis platform. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 463–469. Springer (2011)
9. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 international symposium on Software testing and analysis. pp. 196–206. ACM (2007)
10. Coogan, K., Debray, S.: Equational reasoning on x86 assembly code. In: Proceedings of the 11th IEEE International Working Conference onSource Code Analysis and Manipulation (SCAM). pp. 75–84. IEEE (2011)
11. 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)
12. De Clercq, R., De Keulenaer, R., Coppens, B., Yang, B., Maene, P., De Bosschere, K., Preneel, B., De Sutter, B., Verbauwhede, I.: Sofia: software and control flow integrity architecture. In: Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1172–1177. IEEE (2016)

13. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: Pios: Detecting privacy leaks in ios applications. In: Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS). pp. 177–183 (2011)
14. Fang, H., Zhao, Y., Zang, H., Huang, H.H., Song, Y., Sun, Y., Liu, Z.: Vmguard: an integrity monitoring system for management virtual machines. In: Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS). pp. 67–74. IEEE (2010)
15. Fang, H., Wu, Y., Wang, S., Huang, Y.: Multi-stage binary code obfuscation using improved virtual machine. In: Proceedings of the 14th International Conference on Information Security. pp. 168–181. Springer (2011)
16. Guillot, Y., Gazet, A.: Automatic binary deobfuscation. Journal in computer virology **6**(3), 261–276 (2010)
17. Huang, J., Peng, T.C.: Analysis of x86 instruction set usage for dos/windows applications and its implication on superscalar design. IEICE Transactions on Information and Systems **85**(6), 929–939 (2002)
18. Ibrahim, A.H., Abdelhalim, M., Hussein, H., Fahmy, A.: Analysis of x86 instruction set usage for windows 7 applications. In: Proceedings of the 2nd International Conference on Computer Technology and Development (ICCTD). pp. 511–516. IEEE (2010)
19. Kalysch, A., Götzfried, J., Müller, T.: Vmattack: deobfuscating virtualization-based packed binaries. In: Proceedings of the 12th International Conference on Availability, Reliability and Security. p. 2. ACM (2017)
20. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: Dta++: dynamic taint analysis with targeted control-flow propagation. In: Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS) (2011)
21. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: Proceedings of the 10th ACM conference on Computer and communications security. pp. 272–280. ACM (2003)
22. Kuang, K., Tang, Z., Gong, X., Fang, D., Chen, X., Zhang, H., Wang, Z.: Exploit dynamic data flows to protect software against semantic attacks (2017)
23. Lin, Y., Gao, D., Cheng, X.: Control-flow carrying code. In: Proceedings of the 14th ACM Asia Conference on Information, Computer and Communications Security (AsiaCCS) (2019)
24. Martin, K.M.: Everyday cryptography. The Australian Mathematical Society **231**(6) (2012)
25. Maude, T., Maude, D.: Hardware protection against software piracy. Communications of the ACM **27**(9), 950–959 (1984)
26. Ming, J., Xu, D., Jiang, Y., Wu, D.: Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In: Proceedings of the 26th USENIX Security Symposium (2017)
27. Portokalidis, G., Keromytis, A.D.: Fast and practical instruction-set randomization for commodity systems. In: Proceedings of the 26th Annual Computer Security Applications Conference. pp. 41–48. ACM (2010)
28. Quynh, N.A.: Capstone: Next-gen disassembly framework. Black Hat USA (2014)
29. Rico, R.: Proposal of test-bench for the x86 instruction set (16 bits subset). Tech. rep., Technical Report TR-UAH-AUT-GAP-2005-21-en (2005), http://atc2.aut.uah.es/ gap/
30. Rico, R., Pérez, J.I., Frutos, J.A.: The impact of x86 instruction set architecture on superscalar processing. Journal of Systems Architecture **51**(1), 63–77 (2005)
31. Rolles, R.: Unpacking virtualization obfuscators. In: Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT) (2009)

32. Schwartz, R.J.: The design and development of a dynamic program behavior measurement tool for the intel 8086/88. ACM SIGARCH Computer Architecture News **17**(4), 82–94 (1989)
33. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)
34. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: Proceedings of the 30th IEEE Symposium on Security and Privacy (SP). pp. 94–109. IEEE (2009)
35. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS) (2015)
36. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: Proceedings of the 37th IEEE Symposium on Security and Privacy (SP). pp. 138–157. IEEE (2016)
37. Sinha, K., Kemerlis, V.P., Sethumadhavan, S.: Reviving instruction set randomization. In: Proceedings of the 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 21–28. IEEE (2017)
38. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security. pp. 1–25. Springer (2008)
39. Sullivan, D., Arias, O., Gens, D., Davi, L., Sadeghi, A.R., Jin, Y.: Execution integrity with in-place encryption. arXiv preprint arXiv:1703.02698 (2017)
40. Tang, Z., Kuang, K., Wang, L., Xue, C., Gong, X., Chen, X., Fang, D., Liu, J., Wang, Z.: Seead: A semantic-based approach for automatic binary code deobfuscation. In: Proceedings of the 2017 Trustcom/BigDataSE/ICESS. pp. 261–268. IEEE (2017)
41. van der Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C.: A tough call: Mitigating advanced code-reuse attacks at the binary level. In: Proceedings of the 37th IEEE Symposium on Security and Privacy (SP). pp. 934–953. IEEE (2016)
42. Wang, H., Fang, D., Li, G., Yin, X., Zhang, B., Gu, Y.: Nislvmp: Improved virtual machine-based software protection. In: Proceedings of the 9th International Conference on Computational Intelligence and Security (CIS). pp. 479–483. IEEE (2013)
43. Weiser, M.: Program slicing. In: Proceedings of the 5th international conference on Software engineering. pp. 439–449. IEEE Press (1981)
44. Xu, D., Ming, J., Fu, Y., Wu, D.: Vmhunt: A verifiable approach to partially-virtualized binary code simplification. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 442–458. ACM (2018)
45. Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A generic approach to automatic deobfuscation of executable code. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (SP). pp. 674–691. IEEE (2015)
46. Yang, M., Huang, L.: Software protection scheme via nested virtual machine. Journal of Chinese Computer Systems **32**(2), 237–241 (2011)