Combating Control Flow Linearization

Julian Kirsch, Clemens Jonischkeit, Thomas Kittel, Apostolis Zarras, Claudia Eckert

Combating Control Flow Linearization

Julian Kirsch[1] (✉), Clemens Jonischkeit[1], Thomas Kittel[1],
Apostolis Zarras[1], and Claudia Eckert[2]

[1] Technical University of Munich, Munich, Germany
{kirschju,jonischk,kittel,zarras}@sec.in.tum.de
[2] Fraunhofer AISEC, Munich, Germany
claudia.eckert@aisec.fraunhofer.de

Abstract. Piracy is a persistent headache for software companies that try to protect their assets by investing both time and money. Program code obfuscation as a sub-field of software protection is a mechanism widely used toward this direction. However, effectively protecting a program against reverse-engineering and tampering turned out to be a highly non-trivial task that still is subject to ongoing research. Recently, a novel obfuscation technique called *Control Flow Linearization* (CFL) is gaining ground. While existing approaches try to complicate analysis by artificially increasing the control flow of a protected program, CFL takes the exact opposite direction: instead of *increasing* the complexity of the corresponding *Control Flow Graph* (CFG), the discussed obfuscation technique *decreases* the amount of nodes and edges in the CFG. In an extreme case, this means that the obfuscated program degenerates to one singular basic block, while still preserving its original semantics. In this paper, we present the DeMovfuscator, a system that is able to accurately break CFL obfuscation. DeMovfuscator can reconstruct the control flow, making only marginal assumptions about the execution environment of the obfuscated code. We evaluate both the performance and size overhead of CFL as well as the feasibility of our approach to deobfuscation. Overall, we show that even though CFL sounds like an ideal solution that can evade the state of the art deobfuscation approaches, it comes with its own limitations.

1 Introduction

Software protection (i.e., obfuscation) is a technique used to transform code to make it harder for a human to analyze and understand. In an ideal scenario, obfuscated software maintains its original functionality but it becomes impenetrable to reverse engineering. Therefore, obfuscation offers all the necessary protection mechanisms to software authors that want to protect the internal operations of their programs from the prying eyes of reverse engineers. Here, we can define two groups of software authors: (*i*) software vendors who want to protect sensitive and confidential data shipped together with a piece of software and (*ii*) malware authors who want to evade detection by anti-virus scanners or to hinder inspection by security analysts. Both groups seek software obfuscation for their own purposes.

Over the years, there has been proposed a wide range of obfuscation techniques that were mostly focused on hiding the original control flow by artificially increasing complexity [3, 8, 24]. For instance, *O-LLVM* achieves this by employing Control Flow Flattening *(CFF)*, a technique that conceals the execution sequence of basic blocks [16]. This can also be achieved by employing *virtualization-based* obfuscation techniques. A recent example of such an obfuscator is *Matryoshka* [15] which nests multiple layers of virtualization to cloak the functionality of a protected program.

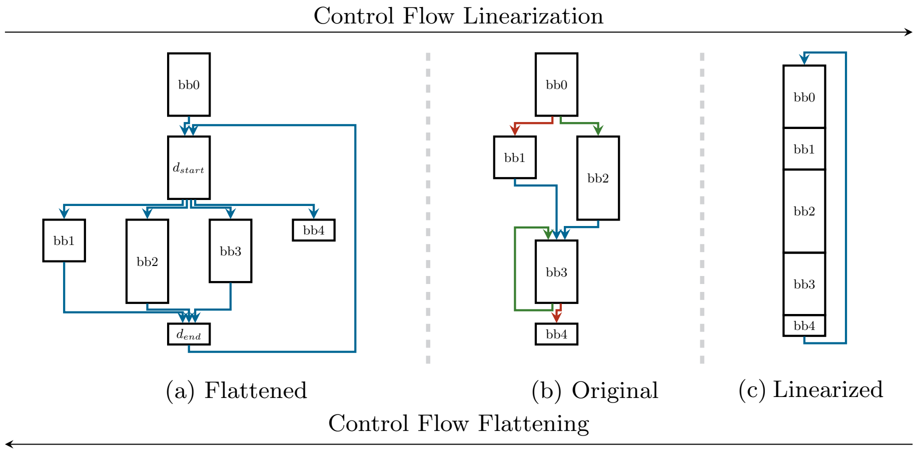
Although obfuscation appears as an optimal solution, it has its own weaknesses. There exist solutions, called deobfuscators, that are centered around a symbolic execution engine and are able to penetrate various obfuscation techniques [27]. KLEE [7] is an example of such a state of the art symbolic execution engine that, however, requires the presence of the source code. ANGR [21,22] and BAP [6] are symbolic execution solutions that do not have a similar requirement. Nevertheless, current approaches for symbolic execution depend on the presence of instructions that explicitly modify the control flow during path enumeration.

Recently, a novel technique, called *Control Flow Linearization* (CFL), makes all control flow changes implicit. In fact, jump free programming is entirely feasible without loosing Turing completeness [11]. CFL constitutes a way of preventing symbolic execution engines from enumerating all satisfiable paths through a program. Therefore, deobfuscation relying on symbolic execution fails to recover the full *Control Flow Graph* (CFG) of a program protected by CFL. This is extremely useful for the software authors that desire to hide the internal operations of their programs. In essence, the MOVFUSCATOR [2], which is to the best of our knowledge the only real world implementation of CFL, helps software to defend itself from reverse engineering.

However, as with any other solution, CFL is not bulletproof. In this paper, we show that it is possible to construct a generic deobfuscator, called DEMOVFUSCATOR, that can reconstruct the control flow, making only marginal assumptions about the execution environment of the obfuscated code. In addition, we evaluate both the performance and size overhead of CFL as well as the feasibility of DEMOVFUSCATOR. Overall, we show that even though CFL sounds like an ideal solution that can evade the state of the art deobfuscation approaches, it is not impenetrable.

In summary, we make the following main contributions:

- We describe the concept of CFL as a novel obfuscation technique and evaluate it in terms of performance and size overhead.
- We propose a generic deobfuscation algorithm to counter CFL and show the effectiveness of our approach.
- We evaluate our approach by recovering the CFGs of various obfuscated binaries, including those of several third-party programs that emerged during past computer security competitions (*Capture-the-Flag contests*).
- We exhibit the advantages of our deobfuscation approach when compared with state of the art symbolic execution techniques.
- We show that CFL, although promising, is far from being perfect.



(a) Flattened

(b) Original

(c) Linearized

Fig. 1: Classification of CFL related to CFF and the original program.

2 Background

In this section, we describe the concepts of CFL and instruction substitution before we shortly describe the Movfuscator [2], an existing example of an obfuscator that implements both CFL and instruction substitution.

2.1 Control Flow Linearization

We call a program *linearized* if it consists of only one singular basic block (excluding initialization of the environment) that ends in a jump targeting itself. Figure 1 shows an example of such a program. CFL makes the control flow of a program implicit by removing all control flow changing instructions without loosing Turing completeness [11].

The central idea of CFL is to duplicate all writable program variables in memory and to re-route all write accesses to either a real set of data or scratch data. This enables the processor to formally execute all instructions in the program while only a subset of instructions affects the current program state. This effect is used to simulate the execution sequence of the basic blocks of the original program and consequently to simulate jumps without the need for branch instructions. In the following paragraphs, we structure this approach to provide a generic transformation strategy to construct linearized programs.

Without the availability of conditional jumps, all instructions of a program need to be executed. We can simulate (un)conditional jumps by mitigating the side effects of memory writes that are not caused by the currently intended basic block. If the effects of the instruction writing to memory should not be visible to the program, we need the code to write to the scratch version of a variable in memory; otherwise the write operation should target the real version of the variable.

Next, we assign a unique LABEL to each basic block of the original program. For simplicity, we use each basic block's virtual address in memory as its LABEL. We also introduce a global STATE variable that during each point in execution holds the LABEL of the *currently executing* basic block. Thus, during execution of the linear program, the program is at all times able to calculate if the current block should write to the real program state or to the SCRATCH version.

Jumps connecting the basic blocks are realized as transitions of the global STATE variable. This is achieved by appending code that updates the STATE variable to each basic block. Note that the STATE variable itself also consists of a REAL and a SCRATCH location, as basic blocks that are not targeted by the current STATE also have to discard their updates to the STATE variable. Based on this construction, any jump predicate can be re-written as the base address of the STATE variable plus the Boolean result of the jump predicate, where TRUE equals 1 and FALSE equals 0. This allows to unconditionally execute each instruction but only if the corresponding predicate is TRUE, the side-effects of the instruction will become visible to the global program state.

Adhering to this construction, it is possible to merge all basic blocks of the original program into one linear basic block without any branch instruction. This effectively mitigates the use of symbolic execution engines to analyze and deobfuscate the generated program as we will see in Section 4. To re-trigger the execution of the program and to give other basic blocks the possibility to execute their payload, a final jump transferring control from the end to the start of the linearized program is appended.

2.2 CFL on the x86 Platform

When implemented for the Intel x86 architecture, CFL faces several challenges. To begin with, while a hypothetical Turing Machine operates on an infinite amount of memory, contemporary von-Neumann systems typically provide only a finite number of addressable bytes in memory. Thus, with finite memory, dereferences of unmapped memory regions can occur if the non-linearized version of the program assigns an invalid value to a index variable (e.g., a pointer in C) *at the global scope*. Even though the dereference with an out-of-bounds index might not be reachable from the point where the new value is assigned in the original program, the linearized version *will* execute the dereference and throw away the side effects later, which might lead to an instant program crash. To mitigate this issue, CFL can be extended to *guard* memory dereferences by adding an instruction that sets dereferenced operands to a known good value if the basic block containing the instruction is *not* active during execution.

Another problem of linearized programs is their ability to call into other functions, as a function call effectively introduces branches into a linearized program. Such high-level primitives can be adopted in two ways. Either the call to a function can be replaced by the called function itself, a process usually referred to as *inlining*, or local variables holding function pointers can be introduced. In the latter case, a variable would point to either the correct call target or to a single `ret` instruction depending on if the basic block containing the call is marked for execution. The two presented ways of handling function calls lead to two different results of the linearization: In the latter case, each function is linearized to one block, whereas in the former case the whole program including all functions is transformed to one block.

2.3 Instruction Substitution

While not directly related to CFL, the MOVFUSCATOR [2] employs instruction substitution to obfuscate a given binary. In context of obfuscation, instruction substitution refers to the process of replacing one or more instructions by an computationally equivalent sequence of instructions that is more difficult to understand for an analyst. Common instructions that are used for instruction substitution are: `ADD`, `ROR`, and `XOR`. Note however, that even restricting substitution to transform to only `MOV` instructions can reach Turing completeness.

Literature discusses instruction substitution in terms of increasing instruction diversity [16] for obfuscation, steganographic applications [13] or in form of case studies of malware [5]. However, *decreasing* the variety of instructions contained in a program is a relatively new idea that, to the best of our knowledge, has first been proposed by Dolan in a formal way [11]. Dolan shows that in an extreme case, instruction substitution can be performed such that the transformed program consists of at most one instruction type: the MOV instruction. We adhere to Domas' terminology and call the process of substituting a program with exclusively MOV instructions *movfuscation*. It is evident that instruction substitution implies CFL, if instructions explicitly changing the control flow are replaced. Both techniques are, however, applicable orthogonally to each other.

2.4 Formalizing the Movfuscator

Using the concepts introduced above, we describe the MOVFUSCATOR, which is to the best of our knowledge, the only public implementation of CFL and instruction substitution. The MOVFUSCATOR is implemented as a compiler back end of the *Little C Compiler* (LCC) [14], capable of compiling programs written in ANSI C. The MOVFUSCATOR is organized as a virtual machine whose instructions are implemented by only MOV instructions.

The MOVFUSCATOR VM consists of four byte-addressable general purpose registers with a machine word size of 32 bits. A stack pointer register points to a full descending stack consisting of 32 bit words. The MOVFUSCATOR VM uses an instruction pointer (IP) that addresses the program at a basic block granularity (we will use the terms IP and TARGET interchangeably). That is, the instruction pointer always points to the beginning of the currently executing basic block. A status register storing comparison results with zero-, signed-, overflow-, and carry-flag works analogously to the x86 status register.

The basic execution is governed by the virtual instruction pointer TARGET and the ON flag. The former contains a LABEL, the virtual address of the basic block that should be executed. It is updated at the end of each basic block of the original program, effectively implementing jump instructions. The ON register is a performance optimization: instead of predicating each memory write access with the result of the comparison LABEL == IP, the comparison is done only at the beginning of each basic block and the result is stored in ON. At the end of each basic block, ON is set to FALSE and TARGET reflects the outgoing edge of the current basic block.

Arithmetic operations are performed by an ALU that is capable of 32 bit integer computations. All computations are performed using look up tables. This constitutes a challenge as the machine word size is equal to the number of addressable bits in the address space. As such, look up tables for all arithmetic and logical instructions grow bigger than the addressable memory space. To circumvent this problem, the inputs for computations are split up into smaller values on which computations are performed using two-dimensional look up tables.

The execution of the generated linearized basic block is re-scheduled infinitely during program execution. To restart execution, the MOVFUSCATOR generates code that transfers the control flow to the beginning of this code. To do so, the code configures itself to be its own, nestable SIGILL handler; execution can be re-triggered at the end of the instruction stream using an illegal mov-instruction.

To interface with the OS, the MOVFUSCATOR follows the application binary interface as defined by external libraries. This means that the obfuscated program sets a special memory location EXTERNAL to the target function's entry in the `plt` section. Afterwards, it prepares the function arguments on the stack pointed to by the `esp` register prior to writing the correct return address on the stack and triggering a segfault by a NULL pointer dereference. This enables the MOVFUSCATOR to call external functions only if execution is enabled by ON. As a matter of fact, there exists a FAULT memory location that contains a valid pointer (no segfault) followed by a NULL pointer that can be accessed similar to other variables. The reason for triggering the segfault is that it provides a mov-only way of directing the execution towards a signal handler (SIGSEGV) that calls the actual library function contained in EXTERNAL.

To prevent generating code with a 1:1 relationship between the original x86 and the MOVFUSCATOR VM's instructions and to defend against pattern recognition, the MOVFUSCATOR employs two hardening techniques. The first is *register shuffling.* Instead of statically assigning registers, the generated code randomly uses one of the EAX, EBX, ECX, EDX general purpose registers for computations. The second is *instruction re-ordering.* The MOVFUSCATOR does a primitive, "overly restrictive" data-dependency analysis on the generated code. This analysis identifies independent pairs of instructions that can be re-ordered without destruction of the program's semantics.

3 Deobfuscation — Control Flow Recovery

In this section we introduce the DEMOVFUSCATOR.[3] Our deobfuscation algorithm is a linear-sweep algorithm that operates in four stages. All assumptions

[3] `https://kirschju.re/demov`

```
mov r1, [b]                     ; α
; instructions not targeting r1
mov r0, [a + r1 * 4]            ; β
```

Fig. 2: Finding SEL_ON

```
mov r0, [sel_on + r1 * 4]   ; γ
; instructions not targeting r0
mov [r0], 1
```

Fig. 3: Usage of SEL_ON

we make are generic for every binary generated by the Movfuscator. Note that while this might seem to be very specific to the Movfuscator, we argue that all obfuscators that implement CFL are per design required to contain similar building blocks. Therefore, our approach is general for different CFL implementations. Our algorithm consists of the four steps introduced in the following.

Finding Key Structures. In this phase we infer the location of *critical data structures* such as the global variable ON indicating whether execution is enabled. Our assumptions are carefully tailored to be applicable to invariants that all linearized programs generated by the Movfuscator satisfy. We also reconstruct the semantic meaning of the respective look up tables that are later used to recover arithmetic computations performed by the code.

Identifying Labels. From instructions that enable execution (i.e., set ON to TRUE), we employ a backward data-flow analysis. Reconstruction of the LABEL is performed by an automatic theorem prover. As a side-effect, this step also reconstructs the location of the global state variable TARGET.

Identifying Jumps and Calls. From instructions that disable execution (i.e., set ON to FALSE), we infer jumps and thus basic block boundaries.

Reconstructing the CFG. Using the gained information, we patch the original binary to make the control flow explicit again.

In the first step, we are required to find critical management data structures of the state machine that were generated by the obfuscator. We first derive the location of the ON data structure from the static initialization code. Note that while a simple pattern matching approach would be sufficient (the static initialization code is approximately the same for all binaries generated by the Movfuscator modulo special compiler flags that omit parts), we improve resilience against changes and further applicability of our approach by reconstructing the location of ON using taint analysis. At a high level, our algorithm determines the location of an instruction that has the shape of instruction β as seen in Figure 2. In the following we write `r{N}` to denote an arbitrary x86 general purpose register. From instruction β, we start a backward taint analysis to infer the origin of register `r1`. If a construction like the above is found, and `b` points to data that has been statically initialized to TRUE, we assume `a` to be the location of SEL_ON, an array whose first entry contains a pointer to the global SCRATCH location and secondly a pointer to ON.

After having identified the location of SEL_ON, we continue by identifying the LABELs of the basic blocks contained in the original program. This is achieved by scanning for an instruction that uses SEL_ON as a base address for an indirect memory access (instruction γ in Figure 3). From this location we employ forward

```
mov r0, [sel_on + r1 * 4]   ; δ
; instructions not targeting r0
mov [r0], 0
```

Fig. 4: Distinguishing conditional and unconditional jumps

```
mov r0, [c + r1 * 4]
; instructions not targeting r0
mov [r0], r2               ; ε
```

Fig. 5: Distinguishing direct and indirect control flow changes

taint analysis to find the point where ON is set to `1` (TRUE). In such a case, we know that the detected instruction is responsible for selecting the ON variable or the SCRATCH variable depending on the result of the predicate stored in `r1`.

We perform a backward taint analysis from γ to reconstruct the predicate that evaluated to the value in `r1`. The backward analysis continues until: (*i*) the beginning of the program is reached, (*ii*) we find another instruction modifying ON, or (*iii*) all taint is sanitized. We then reconstruct the syntax tree of the predicate that evaluates to the truth value contained in `r1` from the tainted instructions. To obtain the original semantic meaning of the operations, we use a-priori knowledge about the look up tables that implement the operations of the MOVFUSCATOR ALU: whenever an instruction accesses a look up table in static memory, we determine the result of the operation for two preselected arguments which are known to evaluate to distinct values for each computation that the virtual ALU is capable of. This approach enables us to reason about the arithmetic and logical operations contained in the reconstructed predicates.

The result of the above step is a Boolean formula that represents the equality check of the basic block's LABEL and the virtual instruction pointer IP indicating whether the current basic block should be executed. From this formula the algorithm obtains the location of the virtual IP as well as the LABEL of the current basic block. The latter is obtained by constraining the predicate to 1 (TRUE) and solving the formula for IP. In this step, our implementation uses the automatic theorem prover z3 [10]. By repeating the above procedure, the algorithm is able to determine the labels of all basic blocks of the program.

In the third step we use the knowledge gained in the second step to evaluate jump and call instructions. Jumps and calls are identified using an approach very similar to the identification of labels. The algorithm performs a second linear sweep and identifies instruction sequences that disable execution by setting ON to `0` (FALSE), which is illustrated in Figure 4. This is needed to determine whether the control flow change is performed conditionally or unconditionally. We use the same technique as explained earlier involving backward taint analysis starting at instruction δ to compute the syntax tree of the predicate contained in `r1`. Using z3 we can decide whether the predicate evaluates to either a constant value, in which case the control flow change occurs unconditionally or alternatively to a formula containing symbolic values, which indicates an conditional jump.

To recover the target basic block label, we need to identify modifications of the virtual IP (instruction ε in Figure 5). In this example, if `r0` is the memory

Predicate Source	**Value Written**	**Recovered Control Flow Change**
Immediate	Constant	Unconditional Direct Jump
Immediate	Formula	Conditional Direct Jump
Stack	Ignored	Return from Call
Other memory	Ignored	Indirect Jump

Table 1: Control flow changes depending on predicate sources and values written.

location of IP then consequently `c` is the memory location of SEL_TARGET, an array holding the global SCRATCH location at index 0 and TARGET at index 1.

After deriving the location of SEL_TARGET we have to distinguish indirect jumps from direct jumps and calls. This is done by analyzing not only the value of `r1` but also the source of the predicate contained in `r2` for each access of SEL_TARGET. Table 1 lists the different decision rules used to determine the type of the control flow change. A basic block never targeted by a jump succeeding an unconditional direct jump is assumed to be a return target. Consequently, we assume the preceeding basic block to end with a call. Note that we do not infer outgoing edges for indirect jump targets, as this is a difficult problem which is heavily discussed by literature. A promising way of resolving indirect jumps is for example using value set analysis [4]. However we want to underline that our algorithm finds the basic blocks that constitute the indirect jump targets.

Following all steps explained above, the algorithm constructs a list of nodes and edges that form the control flow graph of the original program. We use this information to generate images depicting the control flow as well as a patched executable. We do this by ordering all jumps and labels by their respective virtual address and interpreting them as nodes. We iterate over all nodes once. If the current node is a call label, we add an edge to the next element, if it is a conditional jump we add a node in between the current and the next node and add edges between the current and the intermediate node as well as between the intermediate and the next node. In case of an unconditional jump we just add an edge to the target of the particular jump instruction. After this step, all weakly connected nodes form a function and can be merged. By analyzing the calls made from each function, we can then reconstruct the call graph of the analyzed obfuscated binary.

4 Evaluation

To estimate the cost of the obfuscation in terms of size and run-time overhead, we obfuscated three sample programs *Primes*, *Factorial*, and *SHA-256*. Primes is an implementation of the *Sieve of Eratosthenes* calculating all prime numbers smaller than $5 \cdot 10^7$, while Factorial calculates the factorial 20! using a one-dimensional loop. To understand the overhead of programs that are closer to real-world applications, we also evaluated an implementation of the secure hashing algorithm using program SHA-256.

	Primes		**Factorial**		**SHA-256**	
	Non-Lin.	Lin.	Non-Lin.	Lin.	Non-Lin.	Lin.
Non-Sub.	0.88 s	5.03 s	< 0.01 s	< 0.01 s	0.02 s	0.4 s
	240 B	928 B	1884 B	1936 B	5672 B	8564 B
Sub.	62.82 s	289.47 s	< 0.01 s	< 0.01 s	8.09 s	60.57 s
	16,957 B	16,957 B	10,684 B	10,684 B	213,740 B	213,740 B

Table 2: Overhead in terms of run-time (seconds) and code size (bytes).

Primes	**Factorial**	**SHA-256**	**AES**
0.47 s	0.213 s	0.824 s	3.68 s

Table 3: Deobfuscation times of the implementation of our algorithm.

Every program produced eight data points: size and run-time for the non-linearized, non-substituted unobfuscated version as generated by *gcc* version 5.3.1, the linearized and substituted version as generated by the Movfuscator version 2.0 and two versions that were obfuscated using only one of the mechanisms. The linearized, non-substituted version was generated by rewriting the C source code according to Movfuscator while the non-linearized, substituted version is the output of our deobfuscator applied to the movfuscated version. All run times are averaged over ten runs as measured on a Intel Core i7-4770 clocked at 3.4 GHz. For the aforementioned combinations of obfuscation techniques we also added the net size of the generated code in bytes excluding overhead introduced by the executable format. The results can be seen in Table 2.

The measurements show that the linearization itself already leads to a notifiable increase in both run-time overhead and binary size. For example, the SHA-256 program runs about 20 times slower after linearization, while code size increases by roughly a factor of two. This magnitude of overhead makes the obfuscation unsuitable for real-time applications, but could still be used to protect critical parts of an algorithm's implementation. Instruction substitution however leads to a significant overhead both in run-time as well as in binary size. As the calculation of a hash for one megabyte of data takes more than one minute, we argue that this kind of obfuscation is not usable in practice. Note that the size values for the linear and the non-linear version in Table 2 are the same as they differ only by the patched bytes that our deobfuscation algorithm introduced. As relative distances need to remain the same, the size overhead does not change.

To determine the correctness of our deobfuscation algorithm, we compared the CFG of the pre-obfuscated version with the control flow graph of the deobfuscated version of four sample programs: *Primes*, *Factorial*, *AES-128*, and *SHA-256*. Table 3 shows the time required to run our deobfuscation algorithm on the tested binaries. In all cases, except with the simple factorial algorithm, it was faster to deobfuscate the obfuscated binary and to execute the deobfuscated result, than to execute the obfuscated version.

	Clean	**Obfuscated**	**Deobfuscated**
# Basic Blocks Executed	37	99,999	87
Execution Time (s)	5.1	1704.3	17.9
Explored Paths	2	1	3
Executable Size (bytes)	5400	5,962,776	5,962,776

Table 4: Execution times of the angr symbolic execution engine to detect a backdoor in an example executable.

We chose SHA-256 and AES-128 to show that DeMovfuscator works on programs performing complex operations. For AES-128, we followed the official NIST specification on standardized AES vectors and verified that the results of encryption and decryption matched the expected outcomes [12]. To understand the qualitative behavior of our algorithm, we compared the CFG generated from the obfuscated Primes program with its known, unobfuscated C source code. The reconstructed CFG closely matches the original program. This proves that even if a program has been obfuscated with CFL, deobfuscation is still possible.

A good way to show the generality of our approach is to create a pool of binaries, obfuscate them, and then try to reconstruct their CFGs. Internet is obviously the best existing pool to collect binaries. Another source we used to harvest binaries is computer security competitions (Capture-the-Flag contests). These contests often contain clever-crafted binaries which are ideal for our evaluation. To this end, we used both sources and indeed our algorithm was able to reconstruct the control flow for all collected binaries. Our algorithm already became handful in previous Capture-the-Flag contests where it helped us to find an input accepted by the binary and therefore solving the task.

To study the impact of movfuscation on a symbolic execution engine, we reproduced the results of *Firmalice* [21] and measured execution times for the clean, the movfuscated, and the deobfuscated version of the *Fauxware* example backdoor. We used angr from the official repository at commit fe3027 and configuring it to prevent angr from concretizing symbolic memory accesses during the operation of the Movfuscator ALU. As angr currently does not implement the `sigaction` syscall used by the Movfuscator, we adjusted the obfuscated version to call library functions using the PLT rather than the SIGSEGV handler. We also patched out the calls to `sigaction` and replaced the final illegal instruction with a proper jump to re-trigger execution of the basic block. The *Fauxware* executable asks for a username and a password and compares them against a database of legitimate credentials. There also exists an execution path that checks the input against hard coded credentials and thus effectively bypasses the authentication step. To find the existence of the backdoor, the original work proposes to use path exploration to check whether there exists an satisfiable path to the code that should only be reachable for legit users without entering credentials from the user database. We applied the script performing the detection to the original, the obfuscated, and the deobfuscated version of the binary and measured execution times. As Table 4 shows, the backdoor can be found in short time before obfuscation. As the executable is intentionally kept simple, already the second explored path triggers the backdoor condition. Nevertheless, analyzing the same executable in its obfuscated version, ANGR times out after reaching the maximum number of executed basic blocks. Note that even though the MOVFUSCATOR generates code consisting of only one basic block, ANGR counts multiple basic blocks due to the invocation of library functions and a maximum number of instructions that one basic block can contain. Internally the path exploration seems to be unable to reason about symbolic values, as the number of paths (1) shows. We tried to re-run the experiment without a threshold and let it continue for 6 hours without being presented with a result.

After applying our deobfuscation algorithm to the obfuscated binary we let symbolic execution explore the binary and ANGR was able to find the backdoor in less than 20 seconds. One interesting observation is that ANGR needed to explore one additional path. We suppose this to be founded in internal path scheduling discrepancies. The run-time of our deobfuscation algorithm to generate a patched version of this example with reconstructed control flow amounted about 0.18 seconds (averaged over 10 runs).

5 Related Work

The topic of obfuscation to protect software is subject of active research in the academic community. Junod et al. propose *O-LLVM* [16], an obfuscator operating on LLVM intermediate representation. Offering a capricious variety of obfuscation techniques we only highlight *CFF* in context of our work. *CFF* [25] is an obfuscation technique targeting the concealment of a protected software's control flow. Ghosh et al. proposed *Matryoshka* [15], which serves exemplary for the class of *process-level-virtualization* (or *emulation*)-based obfuscation. This obfuscation technique works analogous to *CFF*, except that basic block scheduling is governed dynamically by an arbitrarily chosen byte code of a virtual CPU. Several commercial state of the art obfuscators in the industry such as VMProtect [1], EXECryptor [23], and Themida [17] also employ virtualization-based obfuscation to complicate analysis. It is noteworthy that all of the aforementioned obfuscation techniques aim to increase the complexity of the control flow by inserting additional nodes and edges into the CFG. CFL, on the other hand takes the opposite direction by decreasing the complexity of the CFG. Linear obfuscation was introduced by Wang et al. [26]. The authors propose to obfuscate trigger conditions by using unsolved conjectures such as the Collatz sequence to attack symbolic execution. This concept is orthogonal to CFL and can be combined when the ON or the TARGET registers are read or updated.

The problem of emulation-based obfuscation has been studied for more than a decade. Rolles [18] proposes to use templating languages to generate a compiler that is capable of translating a VMProtect protected sample back to the x86 architecture. Sharif et al. [20] propose a deobfuscation technique for emulators based on execution traces and dynamic taint analysis. Coogan et al. [9] compute the relevance of instructions within an instruction trace based on data-flow towards system calls. This approach allows to further reduce the number of assumptions about the obfuscator used but as a drawback only considers parts of the program covered by the trace. Yadegari et al. [27] overcome this limitation by combining instruction traces with concolic execution [19]. Their results heavily rely on the quality of the symbolic execution engine—an assumption that does not hold for instance for programs obfuscated using the Movfuscator.

6 Conclusion

In this work, we evaluated to the best of our knowledge the only publicly available implementation of CFL. Our evaluation shows that instruction substitution is not applicable in real world scenarios due to its high overhead in terms of execution time and code size. However, the significant overhead and the concealment of explicit control flow changes poses a major challenge to dynamic symbolic execution. We have shown a state of the art symbolic execution engine to fail at path enumeration when analyzing a linearized executable. We have also shown that this problem can be recovered by employing our deobfuscation algorithm and applying symbolic execution to the deobfuscated version. In addition to the run-time overhead, which might be acceptable for the obfuscation of a small but critical part of an algorithm, CFL has a major drawback due to its structure. It depends on the existence of both a block selection register, like the target register within the Movfuscator, and a global on flag governing execution. Our investigation revealed that these registers are relatively easy to detect, as they have to be initialized within the static initialization part of the obfuscated binary and are accessed at the beginning and the end of each basic block of the original program during execution. To harden future CFL implementations the locations of those registers have to be concealed such that static analysis cannot reason about the basic blocks of the program.

Acknowledgements

The research was supported by the German Federal Ministry of Education and Research under grant 16KIS0327 (IUNO).

References

1. VMProtect New-Generation Software Protection. https://vmpsoft.com/.
2. The M/O/Vfuscator. `https://github.com/xoreaxeaxeax/movfuscator`, 2015.
3. D. Aucsmith. Tamper Resistant Software: An Implementation. In *Information Hiding*, 1996.
4. G. Balakrishnan and T. Reps. Analyzing Memory Dccesses in x86 Executables. In *International Conference on Compiler Construction*, 2004.
5. J.-M. Borello and L. Mé. Code Obfuscation Techniques for Metamorphic Viruses. *Journal in Computer Virology*, 4(3), 2008.

6. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *Computer aided verification*, 2011.
7. C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2008.
8. C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical report, 1997.
9. K. Coogan, G. Lu, and S. Debray. Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach. In *Conference on Computer and Communications Security (CCS)*, 2011.
10. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.
11. S. Dolan. Mov Is Turing-Complete. Technical report, 2013.
12. M. Dworkin. Recommendation for Block Cipher Modes of Operation, 2001.
13. R. El-Khalil and A. D. Keromytis. *Hydan: Hiding Information in Program Binaries.* Springer, 2004.
14. C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley, 1995.
15. S. Ghosh, J. D. Hiser, and J. W. Davidson. Matryoshka: Strengthening Software Protection via Nested Virtual Machines. In *International Workshop on Software Protection*, 2015.
16. P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-Llvm: Software Protection for the Masses. In *International Workshop on Software Protection*, 2015.
17. Oreans Technologies. Themida: Advanced Windows Software Protection Systems. http://www.oreans.com/themida.php/.
18. R. Rolles. Unpacking Virtualization Obfuscators. In *Workshop on Offensive Technologies (WOOT)*, 2009.
19. K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *European Symposium on Research in Computer Security*, 2005.
20. M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy*, 2009.
21. Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. v. Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.
22. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State Of) the Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2015.
23. StrongBit Technology. EXECryptor Bulletproof Software Protection. http://www.strongbit.com/execryptor.asp.
24. C. Wang, J. Hill, J. Knight, and J. Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical report, 2000.
25. C. Wang, J. Hill, J. Knight, and J. Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical report, 2000.
26. Z. Wang, J. Ming, C. Jia, and D. Gao. Linear Obfuscation to Combat Symbolic Execution. In *European Symposium on Research in Computer Security*, 2011.
27. B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy*, 2015.