Code Obfuscation against Static and Dynamic Reverse Engineering

Sebastian Schrittwieser[1] and Stefan Katzenbeisser[2]

[1] Vienna University of Technology, Austria
sebastian.schrittwieser@tuwien.ac.at
[2] Darmstadt University of Technology, Germany
katzenbeisser@seceng.informatik.tu-darmstadt.de

Abstract. The process of reverse engineering allows attackers to understand the behavior of software and extract proprietary algorithms and data structures (e.g. cryptographic keys) from it. Code obfuscation is frequently employed to mitigate this risk. However, while most of today's obfuscation methods are targeted against static reverse engineering, where the attacker analyzes the code without actually executing it, they are still insecure against dynamic analysis techniques, where the behavior of the software is inspected at runtime. In this paper, we introduce a novel code obfuscation scheme that applies the concept of software diversification to the control flow graph of the software to enhance its complexity. Our approach aims at making dynamic reverse engineering considerably harder as the information an attacker can retrieve from the analysis of a single run of the program with a certain input, is useless for understanding the program behavior on other inputs. Based on a prototype implementation we show that our approach improves resistance against both static disassembling tools and dynamic reverse engineering at a reasonable performance penalty.

Keywords: code obfuscation, reverse engineering, software protection, diversification

1 Introduction

Today, software is usually distributed in binary form which is, from an attacker's perspective, substantially harder to understand than source code. However, various techniques can be applied for analyzing binary code. The process of reverse engineering aims at restoring a higher-level representation (e.g. assembly code) of software in order to analyze its structure and behavior. In some applications there is a need for software developers to protect their software against reverse engineering. The protection of intellectual property (e.g. proprietary algorithms) contained in software, confidentiality reasons, and copy protection mechanisms are the most important examples. Another important aspect are cryptographic algorithms such as AES. They are designed for scenarios with trusted end-points where encryption and decryption are performed in secure environments and withstand attacks in a black-box context, where an attacker does not have knowledge

of the internal state of the algorithm (such as round keys derived from the symmetric key). In contrast to traditional end-to-end encryption in communications security, where the attacker resides between the trusted end-points, many types of software (e.g. DRM clients), have to withstand attacks in a white-box context where an attacker is able to analyze the software while its execution. This is particularly difficult for software that runs on an untrusted host.

Software obfuscation is a technique to obscure the control flow of software as well as data structures that contain sensitive information and is used to mitigate the threat of reverse engineering. Collberg et al. [8] define an obfuscating transformation τ as a transformation of a program P into a program P' so that P and P' have the same observable behavior. The original program P and the obfuscated program P' must not differ in their functionality to the user (aside from performance losses because of the obfuscating transformation), however, non-visible side effects, like the creation of temporary files are allowed in this loose definition. Another formal concept of software obfuscation was defined by Barak et al. [3]. Although this work shows that a universal obfuscator for any type of software does not exist and perfectly secure software obfuscation is not possible, software obfuscation is still used in commercial systems to "raise the bar" for attackers. In the context of Digital Rights Management systems it is the prime candidate for the protection against attackers who have full access to the client software. While the research community developed a vast number of obfuscation schemes (see e.g. [5] and [16]) targeted against static reverse engineering, where the structure of the software is analyzed without actually executing it, they are still insecure against dynamic analysis techniques, which execute the program in a debugger or virtual machine and inspect its behavior.

In this work we introduce a novel code obfuscation technique that effectively prevents static reverse engineering and limits the impact of dynamic analysis. Technically, we apply the concept of code diversification to enhance the complexity of the software to be analyzed. Diversification was used in the past to prevent "class breaks", so that a crack developed for one instance of a program will most likely not run on another instance and thus each copy of the software needs to be attacked independently. In this work we use diversification for the first time for a different purpose, namely increasing the resistance against dynamic analysis.

The main contribution of the paper is a novel code obfuscation scheme that provides strong protection against automated static reverse engineering and which uses the concept of software diversification in order to enhance the complexity of dynamic analysis. Note that we do not intend to construct a perfectly secure obfuscation scheme, as dynamic analysis can not be prevented. However, our aim is to make attacks significantly more difficult so that knowledge derived from one run of the software in a virtual machine does not necessarily help in understanding the behavior of the software in runs on other inputs.

The remainder of the paper proceeds as follows. After a short overview of related work (Section 2) we introduce our approach in Section 3. In Section 4 we explain how performance is influenced by our method and evaluate security aspects. Finally, a conclusion is given in Section 5.

2 Related Work

There are a number of publications on software obfuscation and their implementation. A comprehensive taxonomy of obfuscating transformations was introduced in 1997 by Collberg et al. [8]. To measure the effect of an obfuscating transformation, Collberg defined three metrics: *potency*, *resilience* and *cost*. *Potency* describes how much more difficult the obfuscated program P' is to understand for humans. Software complexity metrics (e.g. [6,12,22,11,13,21,19]), which were developed to reduce the complexity of software, can be used to evaluate this rather subjective metric. In contrast to potency that evaluates the strength of the obfuscating transformation against humans, *resilience* defines how well it withstands an attack of an automatic deobfuscator. This metric evaluates both the programmer effort (how much effort is required to develop a deobfuscator) and the deobfuscator effort (the effort of space and time required for the deobfuscator to run). A perfect obfuscating transformation has high potency and resilience values, but low *costs* in terms of additional memory usage and increased execution time. In practice, a trade-off between resilience/potency and costs (computational overhead) has to be made. However, the main problem of measuring an obfuscation technique's strength is that a well-defined level of security does not exist, even though it can make the process of reverse engineering significantly harder and more time consuming. Several other theoretical works on software obfuscation can be found in [17] and [23].

As preventing disassembling is nearly impossible in scenarios where attackers have full control over the host on which the software is running, the common solution is to make the result of disassembling worthless for further static analysis by preventing the reconstruction of the control flow graph. To this end, [16] and [5] use so-called branching functions to obfuscate the targets of CALL instructions: The described methods replace CALL instructions with jumps (JMP) to a generic function (*branching function*), which decides at runtime which function to call. Under the assumption that for a static analyzer the branching function is a black box, the call target is not revealed until the actual execution of the code. This effectively prevents reconstruction of the control flow graph using static analysis. However, the concept of a branching function does not protect against dynamic analysis. An attacker can still run the software on various inputs and observe its behavior. Medou et al. [18] argue that recently proposed software protection models would not withstand attacks that combine static and dynamic analysis techniques. Still, code obfuscation can make dynamic analysis considerably harder.

An attack is called a class break, if it was developed for a single entity, but can easily be extended to break any similar entity. In software, for example, we would speak of a class break if an attacker can not only remove a copy protection mechanism on the software purchased, but also can write a generic patch that removes it from every copy of the software. For software publishers, class breaks are dreaded, because they allow mass distribution of software cracks (e.g. on the Internet) to people who would otherwise not be able to develop cracks themselves. The concept of diversification for preventing class breaks of software

was put forth by Anckaert [1]. An algorithm for automated software diversification was introduced by De Sutter et al. [9]. Their approach uses optimization techniques to generate different, but semantically equivalent, assembly instructions from code sequences. While software diversification is an effective solution (see e.g. [2]), it raises major difficulties in software distribution, because each copy has to be different. There is no efficient way for the distribution of diversified copies via physical media (e.g. DVD), and software updates for diversified software are difficult to distribute as well. Franz [10] proposes a model for the distribution of diversified software on a large scale. The author argues that the increasing popularity of online software delivery makes it feasible to send each user a different version of the software. However, a specific algorithm for the diversification process is not given.

Another approach to protect cryptographic keys embedded in software is the use of White-Box Cryptography (WBC), which attempts to construct a decryption routine that is resistant against a "white-box" attacker, who is able to observe every step of the decryption process. In WBC, the cipher is implemented as a randomized network of key dependent lookup tables. A white-box DES implementation was introduced by Chow et al. [7]. Based on this approach, other white-box implementations of DES and AES have been proposed, but all of them have been broken so far (see e.g. Jabob et al. [14], Wyseur et al. [24] and Billet et al. [4]). Michiels and Gorissen [20] introduce a technique that uses white-box cryptography to make software tamper-resistant. In their approach, the executable code of the software is used in a white-box lookup table for the cryptographic key. Changing the code would result in an invalid key. However, due to the lack of secure WBC implementations, the security of this construction is unclear.

Hardware-based approaches would allow to completely shield the actual execution of code from the attacker. However, this only moves attacks to the tamper resistance of the hardware, while raising new challenges like difficult support for legacy systems and high costs. Therefore, hardware-based software protection is out of scope of this work.

3 Approach

Our approach combines obfuscation techniques against static and dynamic reverse engineering. Within this paper, the term static analysis refers to the process of automated reverse engineering of software without actually executing it. Using a disassembler, an attacker can translate machine code into assembly language, a process that makes machine instructions visible, including ones that modify the control flow such as jumps and calls. This way, the control flow graph of the software can be reconstructed without executing even a single line of code. By inserting indirect jumps that do not reveal their jump target until runtime and utilizing the concept of a branching function we make static control flow reconstruction more difficult.

Employing code obfuscation to prevent static analysis is a first step towards running code securely, even in the presence of attackers who have full access to the host. However, an attacker is still able to perform dynamic analysis of the software by executing it. The process of disassembling and stepping through the code reveals much of its internal structure, even if obfuscating transformations were applied to the code. Preventing dynamic analysis in a software-only approach is not fully possible as an attacker can always record executed instructions, the program's memory, and register values of a single run of the software. However, in our approach we aim at making dynamic analysis considerably harder for the attacker by applying concepts from diversification. In particular, the information an attacker can retrieve from the analysis of a single run of the program with certain inputs is useless for understanding the trace of another input. It thus increases costs for an attacker dramatically, as the attacker needs to run the program many times and collect all information to obtain a complete view of the program. This concept can be considered as diversification of the control flow graph.

3.1 Protection against Static Reverse Engineering

In our approach we borrow the idea of a branching function to statically obfuscate the control flow of the software. While previous implementations replace existing CALL instructions with jumps to the branching function, we split the code into small portions that implement only a few instructions and then jump back to the branching function. While this increases the overhead, it makes the blocks far more complex to understand. Because of the small size of code blocks, they leak only little information: A single code block usually is too small for an attacker to extract useful data without knowing the context the code block is used inside the software. The jump from the branching function to the following code block is indirect, i.e. it does not statically specify the memory address of the jump target, but rather specifies where the jump target's address is located at runtime. Static disassembling results in a huge collection of small code blocks without the information on how to combine them in the correct order to form a valid piece of software.

Figure 1 explains this approach. The assembly code of the software is split into small pieces, which we call *gadgets*. At the end of each gadget we add a jump back to the branching function. At runtime, this function calculates, based on the previously executed gadget, the virtual memory address of the following gadget and jumps there. The calculation of the next jump target should not solely depend on the current gadget, but also on the history of executed gadgets so that without knowing every predecessor of a gadget, an attacker is not able to calculate the address of the following one. We achieve this requirement by assigning a signature to each gadget (see Section 3.3). During runtime, the signatures of executed gadgets are summed up and this sum is used inside the branching function as input parameter for a lookup table that contains the address of the subsequent gadget. Without knowing the signature sum of all predecessors of a gadget, it is hard to calculate the subsequently executed gadget.

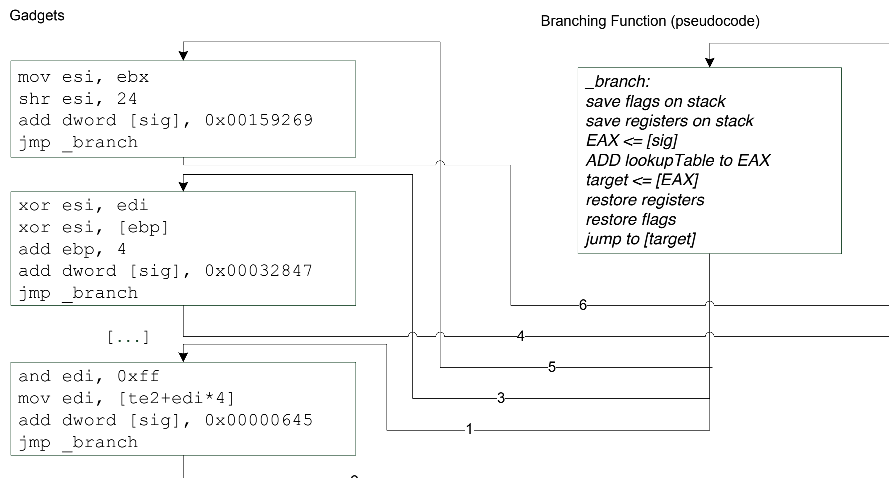

Fig. 1. Overall architecture of the obfuscated program: small code blocks (gadgets) are connected by a branching function.

3.2 Protection against Dynamic Reverse Engineering

The approach effectively prevents static analysis, as a debugger is not able to connect gadgets to each other without calculating signature sums and executing the branching function. Dynamic analysis, however, reveals all gadgets used in a single invocation of the software as well as their order. An attacker can easily remove the jumps to the branching function by just concatenating called gadgets in their correct order. By performing this task for several inputs, he gets significant information on the software behavior.

To mitigate that risk, we diversify the control flow graph of the software so that it contains many more control flow paths than the original implementation. We diversify gadgets (i.e. add semantically identical but syntactical different gadgets to the code) and add input dependent branches so that different gadgets get executed upon running the software with different inputs. We can symbolize this by a gadget graph, where the actual gadget code is stored in the edges that connect two nodes, which symbolize the state of a program. Figure 2 shows the multi-target branching concept before gadget diversification. For every node, we create outgoing edges and fill them with gadgets (i.e. instructions from the original code). All outgoing edges of one node start with the same instruction and only differ in gadget length. In a further step, these gadgets are diversified. Every path through the graph is a valid trace of the program. The branches are input dependent: based on the program's input the branching function decides which path through the graph has to be taken. For a logical connection between gadgets, we implement a path signature algorithm that uniquely identifies the currently executed node and all its predecessors (see Section 3.3).

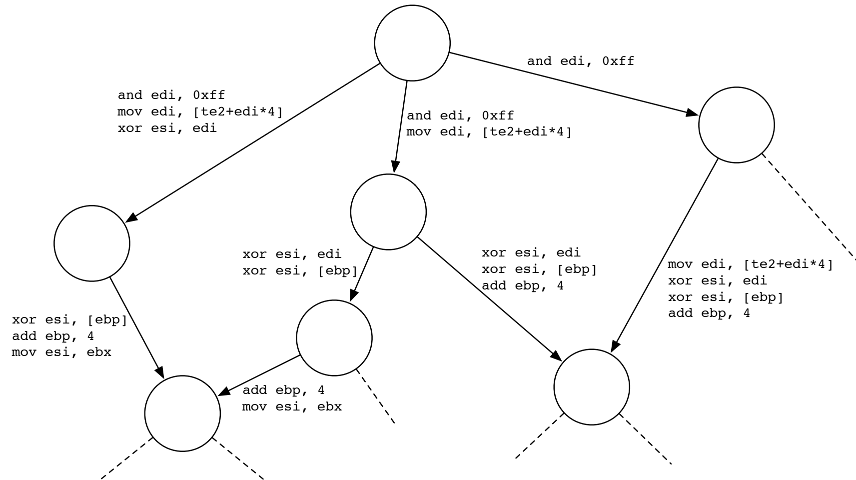

Fig. 2. Gadget graph.

In order to increase the security of the obfuscation, we prevent that a path that is valid for one input is also valid for other inputs. We do this by modifying some instruction's operands and automatically compensate these modifications during runtime by corrective input data. Consider, for example, the assembly instruction `add eax, 8`. If we replace this instruction with `add eax, ebx; sub eax, 1`, where the content of the register `eax` is derived from the program's input, only a value of 9 in `ebx` would yield to the correct value in register `eax`. Figure 3 shows a more complex control flow graph.

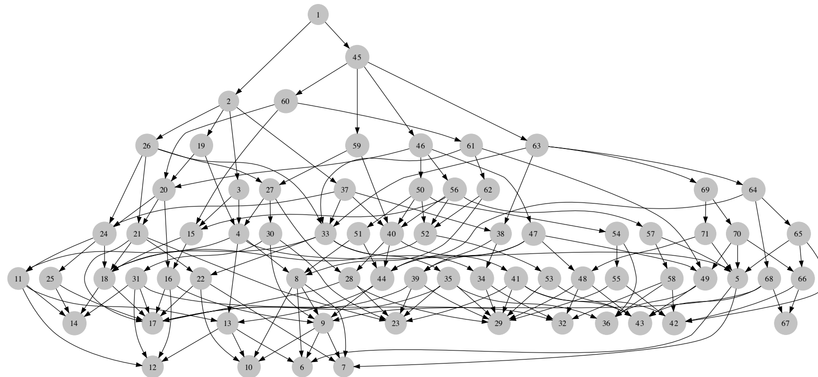

Fig. 3. Diversified control flow graph.

All paths through this graph are valid and semantically equal traces of the program. However, because of the inserted modifications to operands, one specific path yields correct computation only for a specific input (or a group of inputs)

and fails otherwise. If an attacker would use the trace of one input for running the program in the context of another input (e.g. by diverting the control flow in the branching function), our modifications to operands would not be compensated by the new input and the program would show unexpected behavior and might crash at some point (e.g. because of access to miscalculated memory addresses). The process of creating the diversified gadget graph is much easier and faster than breaking the obfuscation as an attacker has to obtain each trace individually.

At the beginning of our obfuscation algorithm, a random gadget graph is created from the software to be obfuscated, based on the input parameters for branching level and gadget size. We then generate unique path signatures (for details see Section 3.3) inside a depth-first search that traverses through all possible paths of the graph. Furthermore, we diversify the gadget code (see Section 3.4), assign the path signature to the gadget and add the gadget to the output file. For every possible path that can be taken to reach a gadget, we add the gadget's memory address and path signature sum to the lookup table. Finally, we attach the branching function and the lookup table to the obfuscated code. Algorithm 1 shows the obfuscation algorithm in pseudocode.

Algorithm 1 Obfuscation algorithm in pseudocode

```
create random gadget graph
DepthFirstSearch (graph)
  while path signature of current gadget is not unique do
    create random path signature
  end while
  diversify gadget code
  add path signature to gadget
  output gadget code
  add gadget's memory address and path signature sum to lookup table
end DepthFirstSearch
output branching function
output lookup table
```

3.3 Graph construction

The main challenge of our approach against dynamic reverse engineering is the performance of the obfuscation algorithm. One the one hand, our approach aims to significantly delay dynamic analysis of an attacker by making it hard to traverse the entire graph within a reasonable time frame (i.e. a brute force attack). However, on the other hand, the initial construction of the graph has to be dramatically less time consuming than an attack. We solve this problem with full knowledge of the structure of the graph at obfuscation time compared to runtime. The obfuscation algorithm creates the graph and stores its structure in memory, allowing very efficient graph traversal at obfuscation time. In contrast, an attacker only has access to the binary code of the software that does not contain an explicit description of the graph's structure. An attacker has to

execute all (or at least most) paths of the graph through the branching function, including the gadget's entire code, in order to rebuild the graph and obtain a complete view of the software.

Our graph construction algorithm takes the original program code as well as a minimum and maximum gadget size and a minimum and maximum branching size as input parameters and is based on a depth-first search. Starting at the root node, the algorithm adds a random number of child nodes (within the bounds of the branching size) and assigns a gadget to each connecting edge. All edges to child nodes contain the same code by means of being filled with a random number of instructions (within the given bounds on the gadget size) from the original code. Only the gadget size and therefore the number of instructions differ at this stage. Gadgets are not diversified at graph construction time. We define the absolute number of instructions executed until reaching a node of the graph as *node level*. Before adding a new node to the graph, the algorithm calculates the node level of the new node and checks if it already exists anywhere in the graph. It that case, instead of creating the node, the algorithm links to the existing node. This method prevents a continually growing width of the graph.

During gadget graph construction, we calculate and store a path signature in each node. We make it unique (see below) so that it clearly identifies the node and all its predecessors. The signature is based on simple ADD and SUB assembly instructions on a fixed memory location. Each gadget adds (or subtracts) a random value to (or from) the value stored in memory. When traversing through the graph, the value stored at the memory location identifies the currently executed gadget and the path that was taken through the graph to reach this gadget. A node can have more than one signature, as more than one path of the graph could reach this node. In that case, each node signature uniquely identifies one of the possible paths from the root to the node. During signature assignment we prevent collisions (two nodes sharing the same signature), by comparing the current signature to all previously calculated signatures and choosing a different value for the ADD or SUB instruction if needed. We decided to implement a trail-and-error approach instead of an algorithm that generates provable distinct signatures to avoid performances bottlenecks at runtime. Figure 4 shows the path signature for a small graph.

We further add a second input parameter to the branching function described in the static part of our approach. Now, both the program's input and the path signature are input parameters for a lookup table that determines the next gadget to be called. To eliminate any information leakage from the branching function's input value, only a hash value of the program's input and the path signature is stored in the lookup table.

3.4 Automatic Gadget Diversification

An efficient generation of semantically equivalent mutations of gadgets is the key challenge for software diversification. This process has to be fully automatic to be able to process large amounts of source code and the transformation function

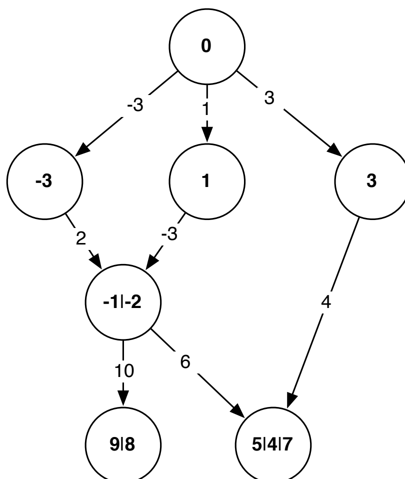

Fig. 4. Path signatures.

is preferably one-way to prevent differential analysis of gadgets. Pattern-based diversification algorithms (e.g. [9]) are a reasonable first code replacement step. However, the fact that an attacker only has local view on a gadget, can help to improve the strength of the diversification by inserting code dependency problems that are locally undecidable for an attacker.

We propose a combination of dummy code insertions and a process we call *instruction splitting*. The idea is to split basic instructions into two ore more instructions that are in combination semantically equivalent to the original instruction and then insert dummy code instructions in between them. We create bogus dependencies between the actual gadget code and dummy instructions by accessing data of split instructions inside the dummy code. To identify and remove dummy instructions, an attacker has to be sure that the code does not perform any vital operations on the code that is executed afterwards. However, this problem is hard to decide due to dependencies between gadgets. Because of the small gadgets sizes, an attacker only has local view on a gadget without knowledge of the subsequently executed gadget.

A simple example is the instruction `add eax, 5` that can be split into the two instructions `add eax, 2` and `add eax, 3`. Of course, this simple transformation provides only very limited security against automatic gadget matching algorithms. We can, however, tremendously improve the strength of the transformation by inserting dummy code. For example, the instruction `mov dword [0x0040EA00], eax` can be considered as dummy code, if the value that is stored in `0x0040EA00` is not used anywhere later in the software. The instruction sequence `add eax, 2; mov dword [0x0040EA00], eax; add eax, 3` is only semantically equivalent to `add eax, 5`, if `mov dword [0x0040EA00], eax` is dummy code. For an attacker with only local knowledge, this is an ambiguous problem.

Simple pattern based transformations do not withstand automated attacks aiming at reversing the diversification. The instructions `test eax, eax` and `cmp eax, 0` are semantically equivalent, but the transformation is weak, because a very simple matching algorithm can easily identify them as equivalent. However, analogous to the instruction splitting method, multi-instruction patterns can be combined with dummy code insertions to enable strong diversification. To provide an example, consider the instructions `push ebp; mov ebp, esp`. A semantically equivalent expression would be `push ebp; push esp; pop ebp`. A simple substitution transformation of one version for the other would most likely not withstand an automated attack. However, if the transformation is combined with dummy code insertion (e.g. `push ebp; push esp; add esp, [0x0040EA00]; pop ebp`, where `0x0040EA00` is `0`), an attacker with local knowledge of the gadget can not reveal the dummy code instructions and hence can not decide gadget equivalence locally.

```
                                xor esi, [ebp]
                                sub ebp, eax
xor esi, [ebp]                  add ebp, 12
add ebp, 4                      add eax, 5
add ebx, 4              τ⇒      add ebx, 2
mov eax, [esp+4]                mov dword [0x0040EA00], ebx
jmp _branch                     add ebx, 2
                                mov eax, [esp+4]
                                jmp _branch
```

Fig. 5. Code block diversification and obfuscation.

Figure 5 shows the transformation of a small code block. The transformation function τ adds dummy code (lines 4 and 6) and modifies the instruction `add ebp, 4` so that it only provides the correct functionality if the corresponding input `8` is loaded into register `eax`. This modification prevents an attacker from extracting this specific (and fully functional) trace and using it with other inputs. To be able to generalize a trace, all input dependent operand modifications would have to be removed, thus the entire code would have to be analyzed instruction by instruction.

4 Discussion

The following section discusses the impact of our obfuscation scheme on performance and size of the resulting program and evaluates security aspects.

Performance and Size. To demonstrate the effectiveness of our approach, we implemented a prototype that reads assembly source code and generates an obfuscated version of it. We measured the performance losses of a simple benchmarking tool as well as a standard AES implementation using 8 different gadgets

sizes. While the dynamic part of our approach accounts for an increase in required memory space because of diversified copies of gadgets, execution time heavily depends on the size and implementation of the branching function, as it inserts additional instructions. The performance decreases with the number of gadgets, due to calls to the branching function, which are required to switch between gadgets. In contrast, the strength of the obfuscation is directly proportional to the number of gadgets, so a trade-off between obfuscation strength and performance has to be made. We compared different gadget sizes from 1 to 50 with the execution times of the non-obfuscated programs (see Figure 6). While very small gadgets result in significant performance decreases, the execution time for a program with a gadget sizes of 10 and bigger approximates the execution time for the original program.

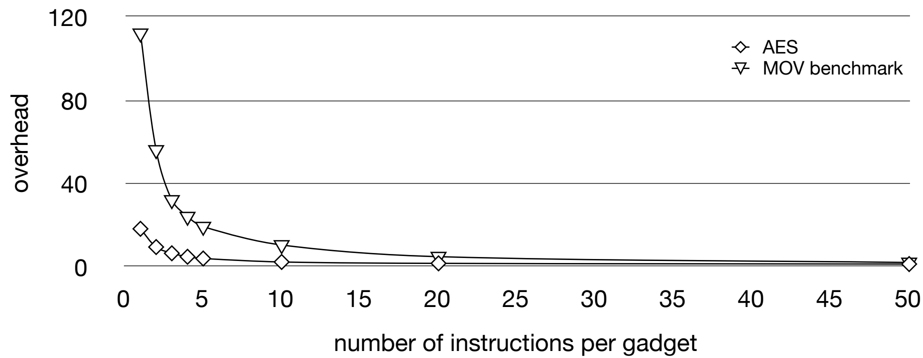

Fig. 6. Execution time for different gadget sizes.

Security. We classified our method with Collberg's metric. Potency (strength against humans) can be evaluated with software complexity metrics. *Program Length* [11], *Nesting Complexity* [12], and *Data Flow Complexity* [22] are increased by our obfuscating transformation and we rate its potency level similar to Collberg's transformation "Parallelize Code" (potency level: *high*). Both methods hide the control flow graph and allow the attacker only local view on small code blocks.

Resilience (strength against automated deobfuscators) is based on the runtime of a deobfuscator and the scope of the obfuscation transformation. The runtime grows *exponentially* with the size of the software and the branching level of the resulting graph, as a deobfuscator has to traverse through the entire graph to reconstruct the control flow. For example, splitting a small program (100 assembly instructions) into gadgets of 12 to 15 instructions and building a gadget graph where every node has 2 to 3 child nodes, yields to more than 1800 different paths through this graph. In Collberg's classification, the scope of

our transformation is "*global*". The combination of both measures results in the resilience level "*strong*".

We furthermore used two state-of-the-art reverse engineering tools to evaluate the strength of the static part of our approach. At first, we tried to reconstruct the program's control flow with the disassembler IDA Pro 5.6. Table 1 compares the automated disassembling rates for the original versions of the code and the obfuscated ones. The values in the table are the percent of successfully reconstructed areas. While IDA Pro was able to reconstruct nearly 38% of the original AES code, the percentage for the obfuscated version declined to about 10%. For the MOV benchmark, the difference was even larger. The results show that for both the AES algorithm and the MOV benchmark, the obfuscated version was much more difficult to reconstruct for IDA Pro. The huge differences between the two examples was caused by different amount of obfuscated code. While for the MOV benchmark the entire code was obfuscated, in the AES example only the algorithm itself was obfuscated. IDA Pro was able to reconstruct non-obfuscated parts of the code correctly, but failed at reconstructing obfuscated code. The disassembler is not able to determine the jump targets of the branching function without actually executing it.

AES algorithm		MOV benchmark	
original	obfuscated	original	obfuscated
37.96%	10.27%	100%	0.13%

Table 1. Amount of successfully reconstructed code areas (IDA Pro).

The second tool we used for evaluation is Jakstab [15] which aims at recovering control flow graphs. Jakstab was not able to resolve the indirect jump at the end of the branching function of our sample program. Although it successfully extracted some of the jump targets from the lookup table, the correct order of the jumps still remained unknown to Jakstab.

Although both tools implement methods for disassembling software and reconstructing control flow graphs, it is not surprising to see them fail at breaking our proposed obfuscation technique as they are not tailored to our particular implementation. Hence, for a more realistic evaluation we also discuss on what a possible deobfuscator for our approach would look like.

One of the main strengths of our approach is that obfuscated software does not contain an explicit representation of the graph structure. It is hidden inside the lookup table, which only reveals the direct successor of a gadget within a single trace during runtime. If an attacker wants to manipulate the software (e.g. remove a copy protection mechanism) he could pursue the following two strategies:

- **Reconstructing the entire graph.** Without obfuscation, an attacker would search for the copy protection code inside the software and then remove it. In our diversified version of the software, however, multiple different ver-

sions of the copy protection are distributed over the entire code. Moreover, they are split into small blocks to fit into the gadgets. An attacker could execute every possible trace of the software and so reconstruct the entire control flow graph. The result would, without doubt, reveal the structure of the code as the individual traces can be analyzed separately. However, the enormous number of possible paths through the graph makes this approach time consuming.
– **Removing diversity of a single trace.** Alternatively, the attacker could remove the copy protection code from one trace and then make this trace valid for all inputs (i.e. remove diversity). The main challenge of this approach is, that the attacker has to analyze and understand the entire trace to be able to identify and remove modifications to operands that were inserted during obfuscation time to bind the code to a specific input.

Neither strategy can likely be performed without human interaction. In the first one, a large number of variants of the same copy protection mechanism would have to be identified and removed manually from the individual traces. In the second strategy, a human deobfuscator would have to analyze an entire trace to be able to identify the inserted modifications that make the trace specific to a single input. We believe, that this high amount of manual effort significantly raises the bar for reverse engineering attacks.

5 Conclusion

This paper proposed a novel software obfuscation method, based on control flow diversification, which makes it difficult for an attacker to relate structural information obtained by running a program several times and logging its trace. By splitting code into small portions (gadgets) before diversification, we achieve a complex control flow graph and static analysis can only reveal very limited local information of the program. We practically evaluated the strength of our approach against automated deobfuscators and showed that it can dramatically increase the effort for an attacker. A performance evaluation showed observable slowdowns for very small gadgets sizes, due to the vast amount of inserted jumps. Versions with bigger gadgets, however, yield to very reasonable performance results.

Future work includes the development of more sophisticated diversification techniques. In contrast to the current implementation where diversification is done only inside gadgets, we consider inter-gadget diversification as an even more effective method against automated gadget matching algorithms.

6 Acknowledgments

The research was funded by FFG - Austrian Research Promotion Agency under grant 826461 (FIT-IT).

References

1. B. Anckaert and K. De Bosschere. Diversity for Software Protection.
2. B. Anckaert, B. De Sutter, and K. De Bosschere. Software piracy prevention through diversity. In *Proceedings of the 4th ACM workshop on Digital rights management*, DRM '04, pages 63–71, New York, NY, USA, 2004. ACM.
3. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology—Crypto 2001*. Springer, 2001.
4. O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *Selected Areas in Cryptography*, pages 227–240. Springer, 2005.
5. J. Cappaert and B. Preneel. A general model for hiding control flow. In *Proceedings of the tenth annual ACM workshop on Digital rights management*. ACM, 2010.
6. S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6), 2002.
7. S. Chow, P. Eisen, H. Johnson, and P. van Oorschot. A white-box DES implementation for DRM applications. *Digital Rights Management*, pages 1–15, 2003.
8. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. 1997.
9. B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. De Bosschere. Instruction set limitation in support of software diversity. *Information Security and Cryptology–ICISC 2008*, pages 152–165, 2009.
10. M. Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*. ACM, 2010.
11. M. Halstead. *Elements of software science*. Elsevier New York, 1977.
12. W. Harrison and K. Magel. A complexity measure based on nesting level. *ACM Sigplan Notices*, 16(3), 1981.
13. S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.
14. M. Jacob, D. Boneh, and E. Felten. Attacking an obfuscated cipher by injecting faults. *Digital Rights Management*, pages 16–31, 2003.
15. J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification*, pages 423–427. Springer, 2008.
16. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003.
17. B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *Advances in Cryptology-EUROCRYPT 2004*. Springer, 2004.
18. M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM workshop on Digital rights management*, pages 75–82. ACM, 2005.
19. T. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 1976.
20. W. Michiels and P. Gorissen. Mechanism for software tamper resistance: an application of white-box cryptography. In *Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 82–89. ACM, 2007.
21. M. Munson Taghi and C. John. Measurement of data structure complexity. *Journal of Systems and Software*, 20(3):217–225, 1993.

22. E. Oviedo. *Control flow, data flow and program complexity.* McGraw-Hill, Inc., 1993.
23. H. Wee. On obfuscating point functions. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing.* ACM, 2005.
24. B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In *Proceedings of the 14th international conference on Selected areas in cryptography*, pages 264–277. Springer-Verlag, 2007.