RUHR-UNIVERSITÄT BOCHUM

Horst Görtz Institute for IT Security

Technical Report TR-HGI-2016-002

Probfuscation: An Obfuscation Approach using Probabilistic Control Flows

Andre Pawlowski, Moritz Contag, Thorsten Holz

Chair for Systems Security

Ruhr-Universität Bochum
Horst Görtz Institute for IT Security
D-44780 Bochum, Germany

TR-HGI-2016-002
1st July 2016

Probfuscation: An Obfuscation Approach using Probabilistic Control Flows

Andre Pawlowski, Moritz Contag, and Thorsten Holz

Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
{firstname.lastname}@ruhr-uni-bochum.de

Abstract. Sensitive parts of a program, such as proprietary algorithms or licensing information, are often protected with the help of code obfuscation techniques. Many obfuscation schemes transform the control flow of the protected program. Typically, the control flow of obfuscated programs is deterministic, i. e., recorded execution traces do not differ for multiple executions using the same input values. An adversary can take advantage of this behavior and create multiple traces to perform analyses on the target program in order to deobfuscate it.
In this paper, we introduce an obfuscation approach which yields *probabilistic control flow* within a given method. That is, for the same input values, multiple execution traces differ, whilst preserving semantics. This effectively renders analyses relying on multiple traces impractical. We have implemented a prototype and applied it to multiple different programs. Our experimental results show that our approach can be used to ensure divergent traces for the same input values and it can significantly improve the resilience against dynamic analysis.

1 Introduction

Obfuscation (lat. *obfuscare* = darken) is the art of disguising a given system such that the analysis becomes harder. In the area of software engineering, obfuscation can be used on either the source code or binary level to obscure the code or data flow. Generally speaking, the goal is to hamper reverse engineering. Code obfuscation plays an important role in practice and such techniques are widely used. On the one hand, obfuscation techniques can be used to protect programs from reverse engineering or to at least increase the costs for such an analysis. Examples include protection systems for sensitive parts or proprietary algorithms of a given program, or digital rights management systems that contain licensing information. On the other hand, obfuscation is widely used by attackers to impede analysis of malicious software such that antivirus companies have a harder time to analyze new samples. As a result, many different kinds of obfuscation techniques were proposed in the last years (e. g., [1–4]). Note that all obfuscation techniques have one constraint in common: the transformations used to obfuscate the program must ensure that the semantic meaning of the program is not changed.

Current state-of-the-art obfuscation techniques translate the target program's code into custom bytecode [6,7]. This bytecode is generated specifically for the obfuscated program and an interpreter is embedded which handles execution of said bytecode. When analyzed statically, the translation to an unknown instruction set forces an analyst to examine the bytecode interpreter first, before actually reverse engineering the original algorithm. Because obfuscation schemes are often difficult to analyze statically, most deobfuscation approaches make use of dynamic analysis [8–10]. A drawback of current obfuscation techniques is the fact that the control flow does not differ for multiple program executions when using the same input values. Thus, it is easier for an analyst to monitor control flow, which exposes parts of the semantic of the target program. Note that state-of-the-art deobfuscation tools utilize a dynamic trace of the program to reconstruct an unobfuscated version of the program.

In this paper, we propose a novel obfuscation approach that tackles the aforementioned problem. Our obfuscation scheme is constructed in such a way that multiple traces of the same function with the same input values lead to different *observed* control flows, whilst preserving semantics. Our approach is inspired by the idea of Collberg et al. [11], which uses opaque predicates constructed using a specifically crafted graph data structure. However, their technique is based on a problem that is only difficult to tackle when the attacker is limited to static analysis. Hence, if an analyst employs dynamic analyses, she can easily determine the value of an opaque predicate which has been executed in the recorded trace. In an empirical evaluation, we show that our proposed obfuscation approach successfully introduces probabilism to the control flow of the target program. Thus, it thwarts dynamic analysis operating on multiple executions of the protected program significantly and does not focus solely on static analysis like other state-of-the-art obfuscation approaches [2,3,6,7].

In summary, we make the following contributions:

– We present a novel obfuscation scheme that introduces probabilistic control flow, but still ensures that the code's semantics are preserved. Due the probabilistic nature of our scheme, it can withstand proposed deobfuscation approaches that rely on a trace-based analysis of several execution runs.
– We implemented a proof-of-concept obfuscation tool in the managed code programming language C# targeting .NET applications. The tool is freely available at `https://github.com/RUB-SysSec/Probfuscator`.
– We evaluate the prototype and demonstrate that probabilistic obfuscation is a viable obfuscation technique to protect sensitive parts of a given program.

This technical report is an extended version of the work presented at DIMVA 2016. This report provides more implementation details and evaluation results.

2 Technical Background

The transformations applied by the obfuscation process aim to hide the program's semantics. If successful, the analysis and deobfuscation effort is considerably higher than feasible for an analyst. In the following, we refer to an analyst as *adversary* given that we study an obfuscation algorithm.

The main class of obfuscation schemes, as well as ours, target the control flow of the target program since it contains vital information about the general structure of a program and exposes high-level constructs such as loops or if-clauses. Doing so, these obfuscation schemes thwart attempts to statically analyze the target program. One building block used by said schemes is the construct of *opaque predicates* [11]. An opaque predicate is a boolean expression whose value is known at obfuscation time. However, its value is difficult to infer by an (automated) attacker. Collberg et al. introduce three types of opaque predicates which we will refer to as *true opaque predicates*, *false opaque predicates*, and *random opaque predicates*, whose expressions evaluate to the boolean values *true*, *false* or evaluate randomly to either, respectively [11]. In the following, we will denote by *(always) taken branch* the branch of an opaque predicate which is known to be always taken.

In case of a true opaque predicate, its taken branch will always be taken, as it corresponds to the predicate evaluating to true. Its other branch also has to point to meaningful code, though, and points to a block of *dead code*. From the obfuscator's point of view, it should be difficult to distinguish dead from live code. False opaque predicates operate analogously. Random opaque predicates differ in that their expression yields a random value and both branches may be taken. Consequently, the code blocks the branches point to have to be semantically equivalent for the obfuscation to be semantics-preserving. A resilient random opaque predicate aims to hide this fact by employing several transformations on the blocks to make comparison of their semantics harder.

Attacks against opaque predicates make use of *data flow analysis* and try to prove that the expression the predicate checks are in fact constant. More resilient opaque predicates hence build expressions involving *pointer aliases* by making use of the hardness of the *intraprocedural may-alias analysis problem* [12]. This problem states that it is generally undecidable if two given pointers into a complex data structure alias each other, i. e., point to the same location in the structure. While algorithms that tackle the problem do exist, many of them are incapable of handling special cases like recursive or cyclic data structures [11].

3 Adversary Model

The goal of the adversary is to analyze and understand a protected algorithm inside the obfuscated method (e. g., a serial key check algorithm or a proprietary algorithm embedded in the method). To this end, the adversary has to understand the effect of the input values on the program's observable behavior, among others. We assume an adversary that bases her deobfuscation attempts solely on dynamic analysis techniques, a common attacker model found in recent literature on attacks against obfuscation schemes [8–10].

The adversary is able to record multiple traces of the obfuscated method for any inputs as well as set breakpoints on specific points in the control flow.

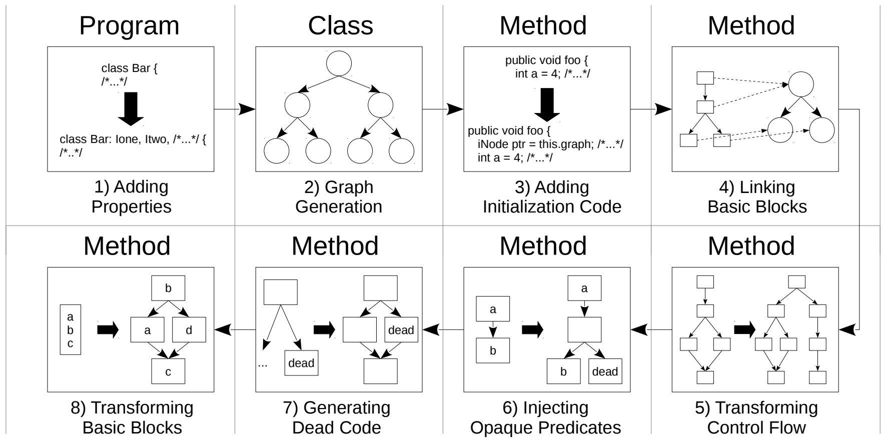

Fig. 1. Overview of the eight steps of the obfuscation process. On the top, it is noted which entity is targeted by the current obfuscation step.

Note that deobfuscation with the help of static analysis is already tackled by obfuscation techniques proposed previously [11–14], which are orthogonal to our approach. However, the adversary is subject to time constraints in her analysis. Given that modern programs change their protection implementations with the release of new versions (e. g., anti-cheat systems, [15]) and recent deobfuscation approaches work solely on execution traces [8–10], we deem these assumptions reasonable.

4 Approach

We propose a novel obfuscation method based on opaque predicates that yields probabilistic control flow whilst preserving semantics. It can be applied to protect sensitive algorithms from dynamic analyses performed by an adversary, thus thwarting recently proposed deobfuscation approaches [8–10]. Our approach makes use of an artificial graph, called *obfuscation graph*, whose nodes consist of objects of classes provided by the target program. Each protected method in the target program holds a pointer to the graph, linking both together. Each basic block of the protected method is linked to one or multiple nodes in the obfuscation graph. During the execution of the protected method, the pointer to the obfuscation graph is moved from node to node. The obfuscation only forwards the pointer to nodes linked to the basic blocks which are to be executed next. With the help of opaque predicates, the scheme ensures that tampering with the link most likely results in a crash of the program.

The obfuscation scheme consists of eight steps which are illustrated in Figure 1 and shortly described in the following.

1. *Adding properties.* The scheme uses properties of the nodes in the obfuscation graph for opaque predicates. In order to increase the number of possible opaque predicates, additional properties are added to the nodes.
2. *Generating the obfuscation graph.* The obfuscator then builds the obfuscation graph with the help of the properties. It is then added to the class that contains the method that should be protected.
3. *Adding initialization code.* This step adds additional logic to initialize the obfuscation scheme for all methods that are to be protected.
4. *Linking basic blocks.* The basic blocks of the control flow graph (CFG) are linked to the nodes of the obfuscation graph. This connection is needed to ensure correct evaluation of the boolean expressions of the opaque predicates.
5. *Transforming control flow.* The CFG of the method is transformed with the help of the linked obfuscation graph in such a way that multiple paths through the CFG yield the same output.
6. *Injecting opaque predicates.* Opaque predicates are injected that only evaluate correctly if the pointer to the obfuscation graph points to the correct location during the execution.
7. *Generating dead code.* Dead basic blocks added during the insertion of opaque predicates are filled with artificially created code.
8. *Transforming basic blocks.* The basic blocks themselves are transformed to obfuscate the method's original code.

In the following, the eight steps are described in detail.

Adding Properties. In order to provide a diverse range of opaque predicates for the same node, the nodes should either have a large number of properties or a property which allows a wide range of different states. Note that all nodes in the obfuscation graph have to implement the *same* properties, which may be uncommon for a set of entities in non-obfuscated applications. Therefore, the obfuscator adds a set of random properties to all possible nodes of the obfuscation graph (i. e., to all classes, as a node is an object of a class). However, the random properties use *different* states.

For our obfuscation approach, a *property* can be anything that can be added to all nodes of the obfuscation graph and can hold different *states*, so that boolean expressions for opaque predicates can be built. For example, common attributes or metadata of a class, like implemented interfaces, can be used. The state of an interface would be a boolean variable indicating whether the class implements the interface.

Generating the Obfuscation Graph. The obfuscation graph is embedded into the class that contains the method(s) that should be protected. If multiple methods of the same class should be protected, the same obfuscation graph can be used multiple times. The nodes of the graph consist of objects of different classes of the target program. Hence, every node is related to a specific class of the program and therefore has different states for the added properties. The

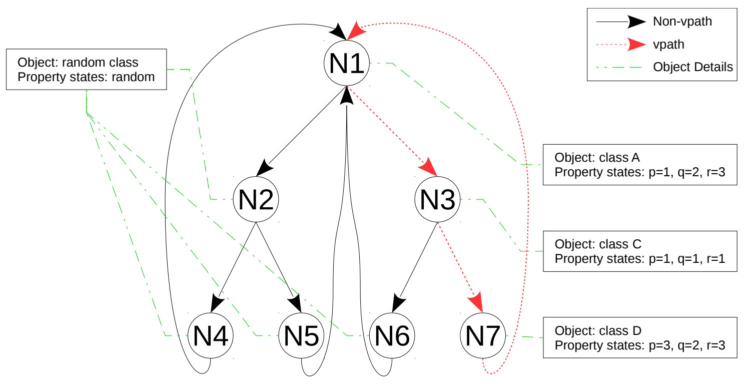

Fig. 2. An example obfuscation graph with one vpath highlighted in red. All classes for the nodes are picked randomly by the obfuscator. The classes and properties that are used for the nodes on the vpath are used to build opaque predicates.

graph is a tree-like graph structure where the leaf nodes have back-edges to the root of the "tree" (semi-cyclic structure).

The structure of the obfuscation graph allows traversal on multiple paths. The obfuscator chooses random paths through the obfuscation graph and declares them to be *vpaths* (as in *valid paths*). The number of vpaths is given by the user. An example for an obfuscation graph is shown in Figure 2. Classes are randomly assigned to the nodes of the graph. The property states of the nodes on the vpaths are later used to build opaque predicates.

The obfuscation graph is parametrized by its *depth* and *dimension*. The depth specifies the maximum length of a path whereas the dimension specifies the number of children of each node. These parameters can be chosen arbitrarily and determine the obfuscation graph's layout. An evaluation of the effect of chosen parameters is given in Section 6.1.

Adding Initialization Code. Because the opaque predicates use properties of the nodes on the vpaths, each method to protect needs a pointer into the obfuscation graph. In order to be consistent between executions, the pointer has to point to the same starting point each time. Therefore, in the beginning of the method, the pointer is reset to the root node of the graph. This pointer realizes the link between executed basic blocks and the nodes in the obfuscation graph.

Obviously, a single vpath can be easily monitored by an adversary using dynamic analysis. Thus, at least *two* distinct vpaths have to exist in the graph. Probabilistic control flow can then be ensured by letting the obfuscated method determine randomly at runtime which vpath is used. Therefore, a *vpath state* is added to each method which determines the vpath used in current transition. It is initialized randomly in the beginning of the method at runtime.

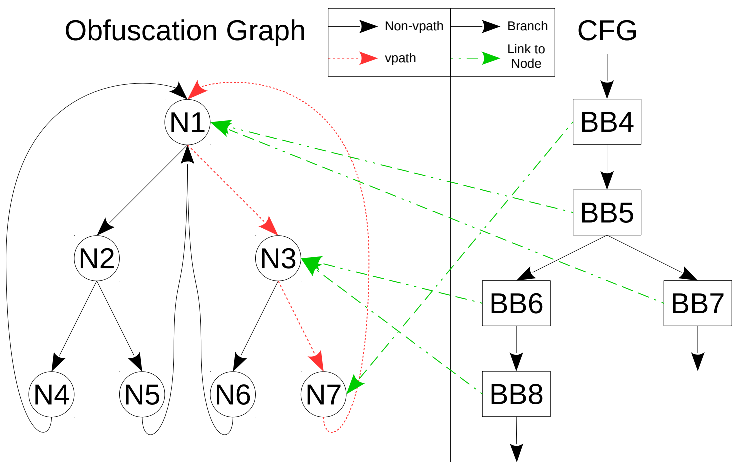

Fig. 3. An example relation between the *obfuscation graph* and the method's control flow. On the right side, a part of the control flow graph is shown. On the left side, the obfuscation graph is shown, where the vpath is highlighted in red. The relation between the nodes of the vpath and the basic blocks is highlighted in green.

Linking Basic Blocks. The nodes on the vpaths are linked to basic blocks in the CFG. Detailed information about the links are used later in the obfuscation process to transform the control flow of the method and to build opaque predicates (e. g., the properties used to construct the opaque predicates). This information is only needed during the obfuscation process. During execution of the method, only the states of the properties are used with the help of opaque predicates to position the pointer into the obfuscation graph. The detailed information is merely kept at obfuscation time.

An example relation of the obfuscation graph and the CFG of the method to protect is shown in Figure 3. The obfuscator links the first basic block of the CFG to the root node of the obfuscation graph (where the *first block* is the one executed first once the method is called). This is the initial position of the pointer into the graph, which is set by the initialization code added previously. The algorithm then iterates over all remaining basic blocks of the CFG and links each basic block to a node on the vpath of the obfuscation graph. During this process, the obfuscator checks for each basic block which node the preceding block is linked to. It then decides randomly to link the current processed basic block to the same node or to the next node on the vpath. This is done for each vpath the obfuscation graph possesses. Hence, each basic block has a link to one node of each vpath. The algorithm terminates when all basic blocks are linked to a node of the obfuscation graph.

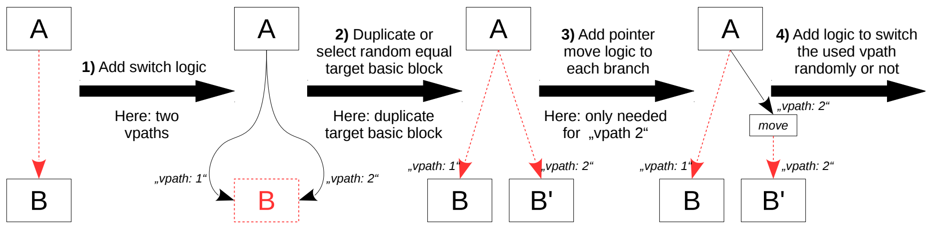
A
B
1) Add switch logic
Here: two vpaths
A
„vpath: 1"
„vpath: 2"
B
2) Duplicate or select random equal target basic block
Here: duplicate target basic block
A
„vpath: 1"
„vpath: 2"
B
B'
3) Add pointer move logic to each branch
Here: only needed for „vpath 2"
A
„vpath: 1"
„vpath: 2"
move
„vpath: 2"
B
B'
4) Add logic to switch the used vpath randomly or not

Fig. 4. The control flow transformation process operating on two consecutive basic blocks A and B. The target of the transformation is highlighted in red. The caption "vpath: X" denotes the control flow path corresponding to the respective vpath in the obfuscation graph.

Transforming Control Flow. The outgoing branches of each basic block are processed exactly once. In the following, we describe the control flow transformation process on the basis of the example shown in Figure 4:

1. Each basic block has a link to one node in every vpath. The *vpath state* (introduced to the protected method while adding the initialization code) determines which of the vpaths is currently active during execution. In order to divert the control flow depending on the currently used vpath, logic must be added that switches the control flow accordingly. Hence, the obfuscator replaces the branch of basic block *A* to *B* with one branch for every existing vpath (in this example there are two vpaths). At runtime, the branch corresponding to the vpath state is taken.
2. In order to avoid all of these new branches having the same target basic block, the obfuscator either duplicates the target basic block or randomly chooses a semantically equivalent basic block. The list of semantically equivalent basic blocks consists of the target basic block itself and all duplicates of this basic block. In this example, the basic block *B* is duplicated and the new basic block *B'* is executed when *vpath 2* is currently active.
3. The source basic block of a branch and the target basic block may be linked to different nodes on the vpath. Hence, the pointer into the obfuscation graph has to be moved from the node the source basic block is linked to to the node the target basic block is linked to (compare Figure 3). As depicted in our example, basic block *B* is linked to the same node on *vpath 1* as basic block *A*, but basic block *B'* is not linked to the same node on *vpath 2* as *A*. Thus, a *move* block has to be inserted in between *A* and *B'*. Said block moves the pointer into the obfuscation graph to point to the node *B'* is linked to.
4. The current approach would not yield probabilistic control flow at all, as the *vpath state* is only set once in the initialization code of a method. Hence, for each outgoing branch of a basic block, logic may be added (determined during the obfuscation process) that may *switch* the vpath the method currently follows. The switching decision is made at runtime and at random. If switching occurs, the pointer into the graph has to be moved according to the chosen vpath.

Injecting Opaque Predicates. In this step of the obfuscation process, the obfuscator adds opaque predicates to the method that should be protected. For each basic block, the obfuscator randomly decides whether to inject an opaque predicate into the incoming branch. If an opaque predicate is injected, the obfuscator randomly decides to either create a true, false, or random opaque predicate. For the true and false opaque predicates, the never taken branch points to a newly created basic block that is marked as *dead*.

During the execution, the method's pointer into the obfuscation graph has to point to the exact node in the active vpath that is linked to the currently executed basic block. For each opaque predicate, the properties that are given by this node are used for its boolean expression. For example, with the obfuscation graph in Figure 2, the obfuscator can build a true opaque predicate for a basic block that is linked to node *N1* with the boolean expression $q == 2$. Note that this boolean expression is not unique to this node in the obfuscation graph, since it is also fulfilled by node *N7* (and probably by other nodes that do not reside on the vpath). This design decision was made to ensure that an attacker is not able to distinctively connect the opaque predicate to a node in the obfuscation graph. Even if the focus of our approach lies on dynamic analysis, the obfuscation scheme should withstand a shallow static analysis.

Furthermore, true and false opaque predicates are deterministic and do not contribute to the probabilism of the control flow. But since the attacker is allowed to conduct a manual dynamic analysis and change the program state during the execution, it adds a tamper proofing mechanism: if the attacker changes the pointer to the obfuscation graph or the obfuscation graph itself in order to affect execution, one of the following opaque predicates would divert the control flow and with a high probability crash the program. This is an advantage over a solely use of random opaque predicates to create probabilistic control flow.

Generating Dead Code. Basic blocks marked as *dead* are filled with artificially generated code. During this process the obfuscator randomly chooses the terminating instruction (called *exit*) of the dead basic block. If the chosen exit is a branch, the target can either be an arbitrary (existing) basic block in the CFG or a new dead basic block. If the target is a new dead basic block, the process is repeated. Otherwise, if the target is an existing basic block, the interconnectivity of the method's CFG is increased.

Transforming Basic Blocks. The transformation of basic blocks is necessary because the algorithm duplicated basic blocks during the control flow transformation step. If no transformation was applied, a pattern matching of basic blocks could be sufficient to detect the always taken branch of an opaque predicate.

In order to make semantically equivalent blocks harder to detect, the obfuscator employs standard obfuscation techniques [16]. We focus on those affecting control flow (like splitting blocks or outsourcing the last instructions to a common block for a subset of blocks), but other techniques can be applied as well.

```
public interface Ibase { /* ... */ }
public interface Itwo : Ibase { /* ... */ }
// ...
foreach(Ibase itemVar in baseItems) {
 if((itemVar as Itwo) != null) {
  /* Do something special with itemVar. */
 }
}
```

Listing 1.1. C# source code depicting a situation where control flow is dependent on implemented interfaces.

This includes instruction re-ordering, replacement of instruction sequences with equal ones, or usage of opaque expressions.

5 Implementation

Our prototype obfuscator is written in C# and targets .NET programs. It uses the *CCI Metadata* libraries [17] in order to transform the target program. For now, the prototype of our obfuscation scheme operates on the bytecode of individual methods a user wishes to protect. In general, however, the approach is not limited to bytecode or methods only (or managed code programming languages). As mentioned in Section 4, the user chooses the method(s) he wants to protect. Note that typically only a very small number of methods in a given software project contain sensitive and valuable information that need to be protected.

We implemented the prototype following the design described in Section 4. All random numbers that are required during the obfuscation process are fetched from the same pseudo random number generator (PRNG). Hence, the seed of the PRNG can be used as a key for the obfuscation. This means the same seed used for the same target method results in the same obfuscated output. In the following, we describe specifics of our implementation.

5.1 Adding Properties

Our approach uses properties that each node in the obfuscation graph possesses. Since we focus on managed code programming languages, we leverage the metadata of the nodes in the obfuscation graph. Specifically, we use the interfaces a class implements. Boolean expressions are created that are used for constructing opaque predicates. A predicate checks a *state* of a property. In this case, whether a node implements a specific interface or not. Since managed code programming languages like CIL or Java allow classes to implement multiple interfaces, opaque predicates can use the same object to check for different interfaces.

The addition of interfaces to classes can cause problems with polymorphic constructs. Hence, our obfuscator adds random self-generated interfaces that do not exist in the non-obfuscated program.

Consider the example given in Listing 1.1. The loop iterates over a list of items named `baseItems` whose items implement the `Ibase` interface. If an item also implements the `Itwo` interface, control flow enters the body of the `if` statement and operates on said item. Now consider the scenario where the obfuscator adds interfaces to all classes of the target program. If the `Itwo` interface is added to a class that is part of the `baseItems` list, control flow will enter the `if` statement's body. However, this control flow branch was not intended by the author of the program and may crash it. Hence, our obfuscator resorts to adding random *self-generated* interfaces.

Still, if an author wishes to add interfaces that look as plausible as possible, our prototype implementation can use already existing interfaces rather than generating them from scratch. Our base set of interfaces consists of all interfaces that are introduced by the target program itself and a list of interfaces provided by the .NET *System* namespace. From this base set, the obfuscator randomly chooses interfaces that are added to the classes of the target program. This can cause problems with polymorphic constructs. To avoid this, the user has to choose the interfaces the obfuscator is allowed to add. Since we consider the user to be the author of the target program, he should know the interfaces that potentially influence control flow.

5.2 Generating Dead Code

The artificially created code that is used for the dead basic blocks has to look as legitimate as possible. If the instructions of a dead basic block do not fit, the block can instantly be recognized as dead block and therefore be ignored during the analysis process. Furthermore, additional obfuscation techniques applied in order to protect against an adversary with static analysis capabilities can lead to difficulties when the dead basic blocks contain illegal instruction sequences.

In order to make the generated dead code as plausible as possible, the dead code generation process can use valid existing basic blocks as *templates*. For example, this is used when generating code for the never taken branch of an opaque predicate. The generator uses the basic block on the always taken branch of the opaque predicate as template. This way, we make sure that both sides of the opaque predicate look plausible, as the other block *is* a legitimate one. Therefore, an analyst must understand the semantics of the basic blocks first in order to distinguish live and dead code in basic blocks.

5.3 Probabilistic Control Flow

The vpath through the obfuscation graph that is used for the current run is randomly determined during execution of the protected method. This randomness is used to implement non-deterministic control flow. We stress that these random numbers are created during the execution of the obfuscated method and not during the obfuscation process.

In our prototype implementation, the random number generator of the .NET *System* namespace is used. This implementation is sufficient for our proof-of-concept tool, but not for a real-world application: an attacker can potentially determine the points in the control flow which generates random numbers and

Table 1. Size of the obfuscation graph and its dependency to the graph's depth and dimension.

Depth	Dim.	|Nodes|	Depth	Dim.	|Nodes|	Depth	Dim.	|Nodes|
6	2	63	7	2	127	8	2	255
6	3	364	7	3	1,093	8	3	3,280
6	4	1,365	7	4	5,461	8	4	21,845
6	5	3,906	7	5	19,531	8	5	97,656
6	6	9,331	7	6	55,987	8	6	335,923

Table 2. Relation between the number of vpaths and the size of the obfuscated method.

vpaths	Basic Blocks	Growth Factor	Branches	Growth Factor
2	304	60.8	357	71.4
3	989	197.8	1,205	241
4	2,520	504	3,059	611.8
5	5,963	1192.6	7,272	1454.4
6	15,418	3083.6	18,804	3760.8
7	26,215	5243	31,848	6369.6

replace them with fixed values. A detailed discussion about the random number generation during the execution of the obfuscated method is provided in Section 7.3. The prototype implementation of our tool is freely available at `https://github.com/RUB-SysSec/Probfuscator`.

6 Evaluation

In this section, we evaluate the prototype of our proposed obfuscation technique. Since it is hard to evaluate obfuscation techniques in general, we evaluate it using the four aspects proposed by Collberg et al. [11]:

1. *Cost* gives a measurement of the time and space overhead that is induced by the obfuscation technique.
2. *Resilience* measures how well the protected program resists deobfuscation attempts.
3. *Potency* measures how complex the program has become after the obfuscation process.
4. *Stealth* measures how well the obfuscation blends into the original program.

Given that our obfuscation is parametrized, we evaluate the effect of the parameters on the obfuscation first. Afterwards, the four aspects cost, resilience, potency, and stealth are measured.

6.1 Obfuscator Parameters

The obfuscation graph is the only component of the obfuscation scheme that is memory dependent. Its size is mainly characterized by its *depth* and *dimension*.

Table 3. Measurable effects of our obfuscation approach

Program	**Size (kB)**		**Avg. Creation (ms)**		**Avg. Comp. (ms)**		**Mem. (kB)**	
	Orig.	**Obfu.**	**Orig.**	**Obfu.**	**Orig.**	**Obfu.**	**Orig.**	**Obfu.**
SHA-256	12.3	7,666.7	< 1	3925	785	5658	1,480	28,852
MD5	13.8	7,068.2	< 1	3924	302	491	1,480	28,880
RC4	9.2	7,058.4	< 1	3917	1209	1842	1,488	28,828

Each node of the graph is represented by an object of a class in the target program and incurs an overhead dependent on the classes that are instantiated. Table 1 shows the size of the obfuscation graph for a range of parameters.

The length of the *vpath* is determined by the depth of the obfuscation graph. The number of vpaths affects the number of possible control flows of the method for the same input and thus influences the method's size as well. The effect of multiple possible control flows is further evaluated in Section 6.3. Table 2 shows the outcome of the obfuscation process for different numbers of vpaths for the same example method. The original method's CFG consists of five basic blocks and five edges. As evident from the table, the growth of the method's size proceeds exponentially.

While larger values for the parameters yield better protection levels, one has to weigh up the desired protection level with penalties in terms of size and speed. These penalties are evaluated in detail in Section 6.2.

6.2 Measuring Cost

In order to evaluate the cost of the obfuscation scheme on the program, we measure its performance, file size, and memory consumption during execution. These values are compared to the execution of the original, unobfuscated program. The tests were run on an Intel Core i7 870 CPU with 2.93 GHz using Windows 8.1 as operating system (OS). We set the number of *vpaths* through the obfuscation graph to six, the *depth* of the obfuscation graph to seven, and the *dimension* of the obfuscation graph to five. The chosen numbers provide a balance between the penalty introduced by the obfuscation scheme and the protection level that is provided, as described in Section 6.1. Since obfuscation introduces a performance overhead and is therefore usually only used to protect important parts of the program, we evaluate our approach only on the implementation of certain algorithms (representative of any intellectual property one wishes to protect). The results of all test cases are shown in Table 3. Because of its nested loop structure and variable input length, we deem the SHA-256 hash computation as best suited to represent a worst case for our obfuscation scheme in terms of performance penalties. The nested loop structure increases the effect of the probabilistic control flow and therefore slows down the computation. Therefore, we describe this test case in detail in the following.

File Size. To quantify the cost of our obfuscation scheme on the file size, we measure both the file size in bytes and the number of instructions of the target method. In our setting, the size of the original binary is 12,288 bytes and the obfuscated binary has a file size of 7,666,688 bytes. This implies that the obfuscated binary is about 624 times larger than the original binary. The method in the original binary consists of 84 instructions, which represent 17 basic blocks and 21 branches. The obfuscated method contains 197,001 instructions, which represent 42,294 basic blocks and 51,277 branches. Thus, the number of instructions in the obfuscated method is 2,345 times greater compared to those of the original method. Note that, as discussed in Section 6.1, the size of the obfuscated binary highly depends on the parameters chosen for the obfuscator. In order to ensure a variety of possible control flows, the obfuscator has to clone the basic blocks of the target method multiple times. Therefore, our obfuscation scheme also increases the size of the target method multiple times. We stress that the growth of the size is dependent on the target method and not on the entire program. A large program has the same growth as a small program if they implement the same method that is the target of the obfuscation. Nevertheless, due to the available resources of modern devices, we see this growth as acceptable.

Performance. The performance is measured by calculating the SHA-256 hash of a 10 MB file. In order to compensate for outliers, we repeat the calculation 1000 times and calculate the average of the needed time (this is done for each algorithm depicted in Table 3). We take two different timings. First, the time needed for the creation of an object of the obfuscated class is measured, and second the time needed for the actual computation of the hash. During the creation of the object itself, the obfuscation graph is built by the constructor of the class. The creation of the obfuscation graph impacts the overall performance depending on the parameters specified by the user. Therefore, we also have to measure the creation and not only the actual computation. Timings are measured with a resolution of 1 ms.

The original binary takes less than 1 ms for object creation. The obfuscated binary takes 3925 ms to create the object (and therefore to build the obfuscation graph). Since we used the same obfuscation parameters for each algorithm tested in Table 3, the time to build the obfuscation graph is nearly identical. The calculation of the hash is performed in 785 ms by the original binary, whereas 5658 ms are needed by the obfuscated binary. While the obfuscated SHA-256 algorithm needs around 7 times more time to perform the same calculation, we stress that this case constitutes a worst case scenario for our obfuscation scheme in terms of performance. The other algorithms shown in Table 3 do not have such a high overhead. Again, these values are dependent on the parameters of the obfuscation graph. While parameters exists for which obfuscation graph creation consumes less time, the protection level for the obfuscated method is lowered as well. Additionally, algorithms that are usually protected with obfuscation in real-world applications are sparsely performed during the execution of a program. Therefore, we regard the introduced performance penalty as acceptable.

Memory. The only memory dependent component of the proposed obfuscation technique is the obfuscation graph. Therefore, the memory consumption of the graph is measured after the object of the protected class is created in the program. The parameters yield an obfuscation graph with 19,531 nodes. The original program consumes 1,480 kB of memory after the object is created. The protected program needs 28,852 kB after the target object is allocated. Therefore, the obfuscation graph needs about 27,372 kB for the used parameters. Note that the needed memory for one obfuscation graph is constant. Since all algorithms in Table 3 are obfuscated with the same obfuscation parameters, they have nearly identical memory consumption. Larger applications embedding the same obfuscation graph will face the same memory requirements. Regarding the available resources of today's devices we see the introduced memory consumption disadvantage as tolerable.

6.3 Measuring Resilience

Resilience measures the resistance of the obfuscation scheme against deobfuscation attempts. Since we focus on thwarting dynamic analysis, we measure the resilience of our obfuscation scheme by quantifying the probabilistic control flow. Therefore, we trace the execution of an obfuscated method with the same input values and compare the similarity of these traces. To this end, we generate a graph from the traced basic blocks in the obfuscated method and compute the *graph-edit distance* between two execution traces using the algorithm proposed by Hu et al. [18]. The graph-edit distance yields the number of edits needed to transform one graph into another graph. Edits are node insertions/deletions and edge insertions/deletions.

We follow the proposal of Chan et al. [19] and normalize the graph-edit distance such that it computes a similarity score using the following formula:

$$\text{similarity}(G_1, G_2) = 1 - \left(\frac{\text{graph-edit distance}}{|G_1| + |G_2|}\right),$$

where the size of the graph G_i is given by the total number of nodes and edges and is denoted as $|G_i|$. The output of the similarity function is a value between 0.0 and 1.0. A result of 1.0 means that the two graphs are identical, whereas a result of 0.0 means they are completely different.

Results. As test case we use our running example, the SHA-256 hash computation. We generated 100 traces by executing the program 100 times in a row with the same input. Unfortunately, the graph-edit distance calculation is NP-hard in general [20]. Therefore, we have to choose an input size that creates traces with graph dimensions that are still comparable. As input data we used 100 bytes of random data. Since the SHA-256 hash computation operates on blocks of 512 bits, the algorithm runs through multiple iterations until it terminates. As obfuscation parameters we use the settings evaluated in Section 6.2.

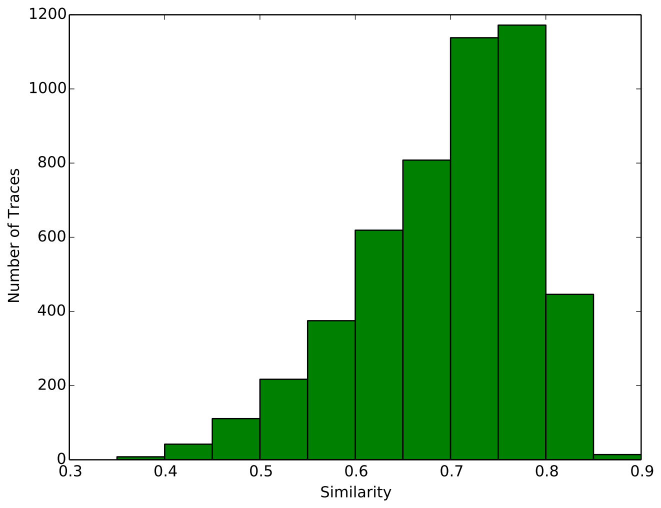

Fig. 5. The 4,950 similarity values of the traces displayed as a histogram. The bin size amounts to 0.05. The smallest similarity was 0.35 and the greatest 0.88. The majority of the values have a similarity of under 0.75.

In total, we calculated 4,950 graph comparisons (as graph comparison of two graphs is commutative). The greatest similarity of two traces was 88.45%. The smallest similarity was 35.29%, while the average of all similarities is 69.65%. An overview of the similarity between the traces is given in Figure 5 as histogram. As can be seen, most of the similarity values are near the calculated average value in the range of 60% to 75%.

The smallest trace regarding the number of unique basic blocks visited 359 unique basic blocks and took 367 unique branches. The largest trace reached 1,183 unique basic blocks and took 1,255 unique branches. To our surprise, the largest trace concerning the number of unique basic blocks is not the largest trace regarding the number of unique branches. On the other hand, the smallest trace regarding the number of unique basic blocks is also the smallest one concerning the number of unique branches. The largest trace regarding the number of unique branches visited 1,178 unique basic blocks and used 1,258 unique branches. On average, 753 unique basic blocks were visited and 793 unique branches were taken by the traces. The number of all visited unique basic blocks and taken unique branches is given in Figure 6. As evident from the figure, the number of visited unique basic blocks and taken unique branches correlate. If more unique basic blocks were executed, more unique branches were used. But still, the number of basic blocks and branches vary greatly between single executions. The size of the traces of the other algorithms is discussed in Appendix A.

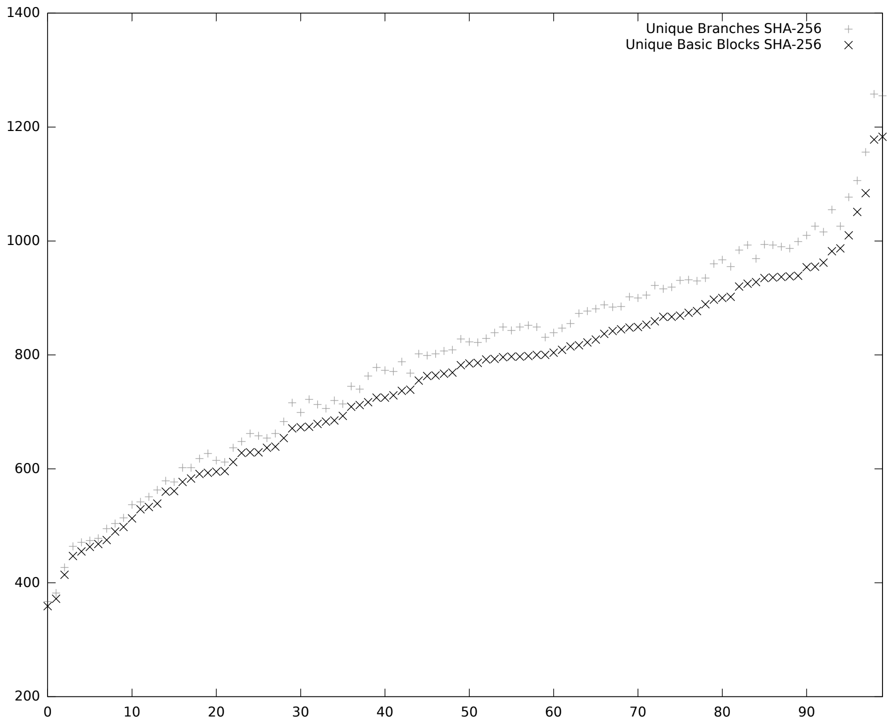

Fig. 6. The number of unique basic blocks and branches each trace used ordered by the number of reached basic blocks. The gray + dots depict the used unique branches and the black x dots show the visited unique basic blocks. On the x-axis the trace number is given. On the y-axis the number of unique basic blocks/unique branches are given.

These results show that multiple executions for the same input values do not even once have the same execution path. This effectively hinders deobfuscation approaches working on multiple traces, such as state-of-the-art deobfuscation methods like [8]. Also, a manual analysis using breakpoints is rendered unreliable in presence of the probabilistic control flow, as we explain in Section 7.1.

6.4 Measuring Potency

Potency measures how complex and confusing the program becomes after obfuscation. In order to evaluate the potency of our obfuscation scheme regarding dynamic analysis, we measure the differences between the original and an obfuscated control flow. Therefore, we recorded an execution trace for the original and the obfuscated program with the same input. During the obfuscation process, the basic blocks of the original method were marked in order to recognize them in the obfuscated CFG. Additionally, all semantically equivalent basic blocks were labeled. This way, we are able to distinguish visited duplicate basic blocks from the original ones. Note that this information is not available for an adversary trying to analyze the obfuscated method.

In order to quantify the *utilization* of the different semantically equivalent basic blocks we visited with respect to all available semantically equivalent basic blocks and the number of executions, we make the following case distinction:

$$\text{utilization} = \begin{cases} \frac{|diff|}{|exec|}, & \text{if } |exec| < |avail| \\ \frac{|diff|}{|avail|}, & otherwise \end{cases},$$

where $|exec|$ gives the number of times one of the semantically equivalent basic blocks were visited, $|avail|$ gives the number of available semantically equivalent basic blocks, and $|diff|$ gives the number of visited different semantically equivalent basic blocks. This way we can differentiate between cases where the total number of visited semantically equivalent basic blocks is lower than the available number of semantically equivalent basic blocks and vice versa. Consider for example a case where only one of the available semantically equivalent basic blocks is executed. If this is the case during multiple iterations of a loop, its utilization of the available semantically equivalent basic blocks is obviously not optimal because control flow visits only this available basic block multiple times. On the other hand, utilization is good if the code contains no loop and control flow visits only one of the semantically equivalent basic blocks during the execution only one single time. Therefore, we have to differentiate between both cases and handle them differently.

Results. Again, we use our running example, the SHA-256 hash computation, as test case. As input data we used 100 bytes of random data and as obfuscation parameters we use the settings evaluated in Section 6.2. We recorded a trace by executing the obfuscated and the original program with the same input. The resulting traces were compared with respect to their executed basic blocks.

The obfuscation process created an obfuscated method which consists of 31,750 basic blocks and 43,698 branches. The obfuscator cloned the basic blocks of the original method multiple times during this process. Remember that the decision if a basic block is cloned is made randomly during the obfuscation process. The minimum number of semantically equivalent basic blocks in the obfuscated method amounted to 9 and the maximum number to 43. On average, the obfuscator created additional 26 semantically equivalent basic blocks for one basic block in the original method. This means that the control flow has on average 27 different possibilities per basic block that exhibit the same behavior.

During the execution of the obfuscated method, the control flow has visited 572 relevant basic blocks that contribute to the calculation of the result. These basic blocks consist of the basic blocks of the original method and transformed copies of these original basic blocks. Of these 572 visited basic blocks, only 24 of them were basic blocks of the original method. Therefore, only 4.2% of the executed relevant basic blocks were the basic blocks of the original method. The utilization of the available semantically equivalent basic blocks range from 12.9% to 100%. In total, 71% of the available semantically equivalent basic blocks were utilized during the execution of the obfuscated method. A detailed overview of the results of all tested algorithms is depicted in Table 4 in Appendix B.

The results show that an execution of the obfuscated method uses a variety of different but semantically equivalent basic blocks to compute its result. Hence, the number of basic blocks that are actually involved in the computation has been increased by our obfuscation scheme and thus the complexity of the control flow.

6.5 Measuring Stealth

Stealth measures the difficulty for an adversary to determine if the given method is obfuscated, i. e., how well the obfuscated entity fits in legitimate code. Although stealth is not an objective of our approach, we evaluate it for the sake of completeness. Recently published obfuscation papers measure this aspect based on the distribution of instructions [21–23]. However, as Collberg et al. [11] describe it, stealth is a *context-sensitive* metric. Hence, instead of pursuing a static approach for evaluating stealth, we consider the *dynamic* behavior of the obfuscated program. This fits our general focus on dynamic analysis.

Given that our approach is by design supposed to yield different execution traces for the same input, *stealth* is inherently hard. An adversary only has to execute the program two times with the same input and compare the recorded execution traces. If they differ, the adversary can conclude that the program is most likely protected by our obfuscation approach. In particular, if she can run methods independently, it is even possible to pinpoint the exact method that has been obfuscated.

7 Discussion

In the following, we discuss potential limitations and shortcomings of our approach.

7.1 Dynamic Analysis

Our approach aims to transform methods such that multiple traces of the same function using the same inputs differ, which implies that dynamic deobfuscation approaches are hampered [9, 10]. Furthermore, this is done to thwart dynamic analyses operating on multiple executions (like [8]). For example, *manual* dynamic analysis of the obfuscated method is hindered by probabilistic control flow: an adversary observing the control flow at some fixed point during execution of the method cannot depend on the program reaching the exactly same point during a following run. Hence, pausing execution using breakpoints is rendered unreliable in presence of our obfuscation approach.

7.2 Single Trace Analysis

If an adversary knows that our obfuscation scheme is used, the best way to circumvent it is by resorting to work on a single execution trace. Since the goal of probabilistic control flow is to make dynamic analyses based on multiple traces harder, deobfuscation methods operating on *one* trace are only affected if a loop is present. We note that algorithms processing the recorded trace dismiss basic blocks that do not affect the outcome of the method [8–10]. However, as shown in Section 6.4, during multiple loop iterations, different semantically equivalent basic blocks can be hit. These blocks may affect the method's outcome and thus complicate the resulting trace. As future work, we propose integrating the obfuscation graph better into the protected method by means of *opaque* values. This way it gets harder to dismiss instructions based on their usage of the obfuscation graph only.

Furthermore, deobfuscation approaches operating on one trace do not perform as good in terms of *code coverage* compared to those using multiple execution paths. This poses a problem for an adversary who wants to analyze multiple execution paths in an algorithmic manner in order to understand the obfuscated program better. Often, *multi-path exploration* techniques are considered when tackling this problem [8, 10]. This is where our approach proves useful: It introduces a variety of valid, but distinct control flows and works probabilistically. For an adversary, it is hard to distinguish whether a branch was taken due to probabilistic control flow or because the function was run with different input. In order to improve this aspect, we currently work on extending our approach by *inlining* the semantics of multiple methods into one method. The semantic that is actually executed when the method is called is then determined with the help of the obfuscation graph and opaque predicates.

7.3 Probabilistic Control Flow

The obfuscation graph and its vpaths are an integral component of our approach. Vpaths are used to select the current control flow through the obfuscated method and therefore to enable probabilistic control flow. Which vpath is to be used is decided randomly. In our prototype implementation, the active vpath is chosen using the PRNG provided by the .NET *System* namespace. Obviously, this implementation is vulnerable, as the call to the PRNG could be replaced by fixed values. This reduces probabilistic control flow to a deterministic one.

A straightforward approach to make the random number generation more resilient is not to use any external PRNG. Instead, one could insert a PRNG into the obfuscated method itself and replace the calls to the external PRNG with code sequences that generate random numbers. This way, the random number generation is harder to pinpoint by an adversary, especially if multiple, diverse sequences are inserted. Furthermore, the random number generation can be hidden beneath additional layers of obfuscation.

However, even this construct suffers from the problem that it requires an initial random seed to create different control flows every time it is executed. If an adversary is able to set this initial random seed to a fixed value, the PRNG in the obfuscated method generates the same sequence of random numbers every time the program is executed. This circumstance poses the greatest limitation of our current implementation. However, due to the huge amount of values that can be fetched from the operating system that can influence the seed, it is not easy in practice to fixate them all. For future work, we propose to develop methods to conceal the retrieval of external states that are used to seed the PRNG and to transfer the selection of used vpaths through the obfuscation graph from an external entity to the program (i. e., a server).

7.4 Opaque Predicates

It is possible to create probabilistic control flow with the help of random opaque predicates. The difficulty resides in creating *resilient* predicates. Our approach uses the idea of Collberg et al. [11] to embed a graph structure which is used to construct resilient opaque predicate based on the hardness of detecting aliases of the pointer into the graph. A further advantage of our obfuscation scheme is that this introduces tamper resistance for the probabilistic control flow.

7.5 Other Programming Languages

Although our proof-of-concept implementation uses metadata available to managed code programming languages, the scheme can also be implemented for other programming languages. In object-oriented programming languages like C++, the obfuscation graph can be realized by using objects of classes for the nodes and attributes of the object as properties for the opaque predicates. Furthermore, our obfuscation approach is also not necessarily limited to object-oriented programming languages either. Generally, our approach requires any program entity usable for a node in the graph that also contains enough properties to build opaque predicates (see Section 4). Hence, adopting our obfuscation approach for other programming languages is merely an engineering effort and not a design limitation.

8 Related Work

The basic technique our approach is based on is presented in a paper by Collberg et al. [11]. They propose a method to create opaque constructs based on objects and pointer aliases. They also suggest a directed graph as concrete data type. However, their approach is mainly concerned with the creation of cheap, stealthy and resilient opaque constructs. We extend this approach and focus on the different paths we can insert into a target using their construct. This stems from the insight that while their technique efficiently makes static analysis harder, the traces obtained using dynamic analyses are very much the same. This, in turn, helps in determining the concrete value of an opaque predicate and might allow to partly reconstruct the control flow of the program.

Wang et al. describe a technique to obfuscate a target program using control flow transformations as well [13]. They transform a method's CFG in such a way that a new basic block in the beginning of the method decides which original basic block is executed next. These control flow decisions are made based on a state variable which gets updated after every basic block. Similar to the approach of Collberg et al., they transform the control flow analysis problem into a data flow analysis problem. However, their approach also merely aims to make static analysis of an obfuscated program harder.

More recent work focuses on deobfuscation of obfuscated programs [8–10]. All of them have in common that they are based on dynamic analysis. Traces of the program's execution are recorded and subsequently used to remove the applied obfuscation schemes. Approaches working on multiple traces in order to tackle the code coverage problem [24] of dynamic analysis are challenged by the probabilistic control flow introduced by our technique.

The recent work of Crane et al. also make use of probabilistic control flow [25]. It enables them to thwart cache side-channel attacks. To this end, they clone program fragments and transform the clone in order to avoid making an exact copy. A stub is used to decide randomly if the clone or the original fragment is executed. Because an attacker has no knowledge about which was executed, it hampers cache side-channel attacks. Additionally, Davi et al. [26] use probabilistic control flow in combination with fine-grained memory randomization in order to prevent conventional return-oriented programming (ROP) and JIT (just-in-time)-ROP attacks. To this end, they clone and diversify the code that is loaded into memory. Whenever a function is called, their system randomly decides if the original or cloned function is executed. Once the executed function returns, the system checks if execution shall continue at the normal or cloned version of the function caller by adding an offset to the return address. Therefore, an attacker is not able to precisely predict where execution will resume and cannot reliably perform a ROP or JIT-ROP attack.

9 Conclusion

In this paper, we introduce a novel approach to obfuscate software, including, but not limited to, those written in managed code programming languages. The proposed scheme is based on a construct introduced by Collberg et al. [11]. However, instead of focusing on protecting the program against static analysis, we introduce a scheme achieving probabilistic control flow, aiming to make dynamic analysis harder. This is achieved by embedding an obfuscation graph containing multiple virtual paths. Based on these paths, opaque predicates are constructed and added to the target method. Consequently, control flow may take different paths exhibiting the same observable semantics.

We have implemented a prototype obfuscator for .NET applications and evaluated it using multiple programs. The experiments have shown that the obfuscated methods do not exhibit the same execution trace after executing it 100 times in a row with the same input. Inevitably, this comes with a significant performance and memory penalty. Resilience against dynamic analyses thus has to be weighed up with constraints on time and space. We are confident that the overhead is still acceptable to protect sensitive parts or proprietary algorithms of a given program. Since we believe our obfuscation approach provides a new strategy for tackling dynamic analysis and hence a building block for future research, we are making our obfuscation tool available to the research community as open source software.

References

1. B. Lee, Y. Kim, and J. Kim, "binOb+: A Framework for Potent and Stealthy Binary Obfuscation," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2010, pp. 271–281.
2. Collberg, Christian, "The Tigress C Diversifier/Obfuscator," http://tigress.cs.arizona.edu.
3. Junod, Pascal, "Obfuscator-LLVM," https://github.com/obfuscator-llvm/obfuscator/wiki.
4. H. Fang, Y. Wu, S. Wang, and Y. Huang, "Multi-stage Binary Code Obfuscation using Improved Virtual Machine," in *Information Security*. Springer, 2011, pp. 168–181.
5. Oreans Technologies, "Themida: Advanced Windows Software Protection System," http://www.oreans.com/themida.php.
6. ——, "Code Virtualizer: Total Obfuscation against Reverse Engineering," http://oreans.com/codevirtualizer.php.
7. VMProtect Software, "VMProtect: Software protection against reversing and cracking," http://vmpsoft.com/.
8. 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 (S&P)*, 2015.
9. K. Coogan, G. Lu, and S. Debray, "Deobfuscation of Virtualization-obfuscated Software: a Semantics-based Approach," in *ACM Conference on Computer and Communications Security (CCS)*, 2011, pp. 275–284.
10. M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic Reverse Engineering of Malware Emulators," in *IEEE Symposium on Security and Privacy (S&P)*, 2009, pp. 94–109.
11. C. Collberg, C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in *ACM Symposium on Principles of Programming Languages (POPL)*, 1998, pp. 184–196.
12. G. Ramalingam, "The Undecidability of Aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.
13. C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of Software-Based Survivability Mechanisms," in *International Conference on Dependable Systems and Networks, 2001. DSN 2001.* IEEE, 2001, pp. 193–202.
14. B. Anckaert, M. Jakubowski, and R. Venkatesan, "Proteus: Virtualization for Diversified Tamper-Resistance," in *Proceedings of the ACM workshop on Digital rights management.* ACM, 2006, pp. 47–58.
15. Kushner, David, "Steamed: Valve Software Battles Video-game Cheaters," http://spectrum.ieee.org/consumer-electronics/gaming/steamed-valve-software-battles-videogame-cheaters.

16. C. Collberg, C. Thomborson, and D. Low, “A Taxonomy of Obfuscating Transformations,” Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.
17. Guy_Smith, “Common Compiler Infrastructure: Metadata API,” https://ccimetadata.codeplex.com/.
18. X. Hu, T.-c. Chiueh, and K. G. Shin, “Large-scale Malware Indexing Using Function-Call Graphs,” in *ACM Conference on Computer and Communications Security (CCS)*, 2009, pp. 611–620.
19. P. P. Chan and C. Collberg, “A Method to Evaluate CFG Comparison Algorithms,” in *International Conference on Quality Software (QSIC)*, 2014, pp. 95–104.
20. Z. Zeng, A. K. Tung, J. Wang, J. Feng, and L. Zhou, “Comparing Stars: On Approximating Graph Edit Distance,” in *International Conference on Very Large Data Bases (VLDB)*, 2009.
21. H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-c. Yew, “Control Flow Obfuscation with Information Flow Tracking,” in *Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.
22. P. Wang, S. Wang, J. Ming, Y. Jiang, and D. Wu, “Translingual Obfuscation,” in *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.
23. I. V. Popov, S. K. Debray, and G. R. Andrews, “Binary Obfuscation Using Signals,” in *USENIX Security Symposium*, 2007.
24. A. Moser, C. Kruegel, and E. Kirda, “Exploring Multiple Execution Paths for Malware Analysis,” in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2007, pp. 231–245.
25. S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, “Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity,” in *Symposium on Network and Distributed System Security (NDSS)*, 2015.
26. L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, “Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming,” in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

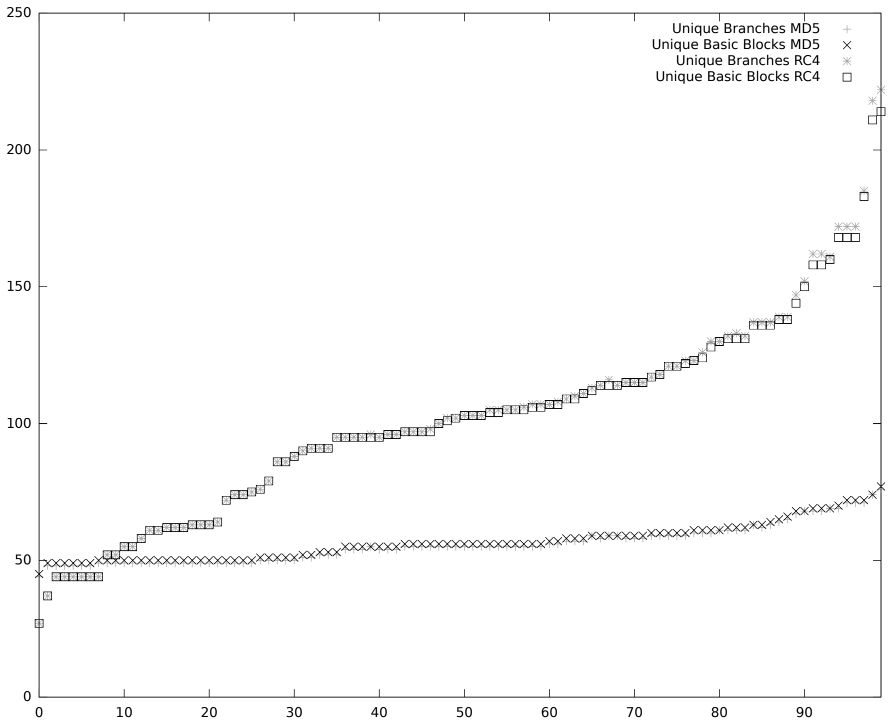

Fig. 7. The number of unique basic blocks and branches each trace used ordered by the number of reached basic blocks. The gray dots depict the used unique branches and the black dots show the visited unique basic blocks. On the x-axis the trace number is given. On the y-axis the number of unique basic blocks/unique branches are given.

A Probabilistic Control Flow

Figure 7 shows the number of unique basic blocks visited and unique branches taken for 100 traces with the same input for the obfuscated MD5 and RC4 algorithm. Like the result of the traces for the SHA-256 algorithm (evaluated in Section 6.3), the number of visited unique basic blocks and taken unique branches correlate. If more unique basic blocks were visited, more unique branches were taken during the execution of the obfuscated method. For the MD5 algorithm, the smallest trace regarding the number of unique basic blocks reached 45 unique basic blocks and took 44 unique branches. This is also the smallest trace regarding the number of unique branches. The largest trace regarding the number of unique basic blocks visited 77 unique basic blocks and used 76 unique branches. This is also the largest trace regarding the number of unique branches. On average, each trace used 56 unique basic blocks and 55 unique branches. For the obfuscated RC4 algorithm, the smallest trace regarding the number of unique basic blocks reached 27 unique basic blocks and took 27 unique branches. This

Table 4. The results of the comparison of the obfuscated method trace with the trace of the original method for the same input (ID = ID for semantically equivalent basic blocks, $|avail|$ = number of available semantically equivalent basic blocks, $|exec|$ = total number of times one of the semantically equivalent basic blocks were visited, $|diff|$ = number of different semantically equivalent basic blocks executed, $|orig|$ = number of times the basic block of the original method was visited instead of a semantically equivalent basic block, *Util.* = utilization of the reached different semantically equivalent basic blocks with respect to available semantically equivalent basic blocks and the total number of executions in percent).

ID	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**Total**
$\|avail\|$	9	43	40	30	35	24	22	20	29	18	22	31	25	22	43	23	33	469
$\|exec\|$	1	20	1	19	3	1	2	1	34	2	32	98	2	96	130	2	128	572
$\|diff\|$	1	10	1	8	3	1	2	1	15	1	10	4	1	4	24	2	20	108
$\|orig\|$	0	1	0	6	0	0	0	0	0	0	0	0	0	0	6	0	11	24
Util.	100	50	100	42.1	100	100	100	100	51.7	50	45.5	12.9	50	18.2	55.8	100	60.6	71

SHA-256

ID	**0**	**1**	**2**	**3**	**4**	**Total**
$\|avail\|$	6	33	27	20	17	103
$\|exec\|$	1	3	1	2	1	8
$\|diff\|$	1	3	1	2	1	8
$\|orig\|$	0	0	0	0	0	0
Util.	100	100	100	100	100	100

MD5

ID	**0**	**1**	**2**	**3**	**4**	**Total**
$\|avail\|$	11	34	18	24	18	105
$\|exec\|$	1	101	1	100	1	204
$\|diff\|$	1	17	1	12	1	32
$\|orig\|$	0	3	0	3	0	6
Util.	100	50	100	50	100	80

RC4

is also the smallest trace regarding the number of unique branches. The largest trace regarding the number of unique basic blocks and unique branches visited 214 unique basic blocks and used 222 unique branches. On average, 101 unique basic blocks and 102 unique branches were used by the traces.

As evident from the figure, the numbers for the MD5 algorithm do not have such a wide range like the obfuscated RC4 algorithm (same figure) or the SHA-256 algorithm (Figure 6 in Section 6.3). This can be explained by the small number of iterations the loop in the algorithm passes through with the given input. The other algorithms have a larger number of iterations during their execution (the actual numbers are shown in Table 4 of Appendix B).

B Basic Block Utilization

In Table 4, a detailed overview of the results of the comparison of the obfuscated and original trace is given. The semantically equivalent basic block utilization for the SHA-256 algorithm ranged from 12.9% to 100%. In total, 71% of the available semantically equivalent basic blocks were utilized. For the MD5 algorithm, the utilization always reached 100%. This high utilization is explained by the fact that the algorithm does not have many iterations for the used input. Therefore, there is a high probability that always different semantically equivalent basic blocks are used during the execution. The utilization for the RC4 algorithm ranged from 50% to 100%. In total, 80% of the semantically equivalent basic blocks were utilized.