VOT4CS: A Virtualization Obfuscation Tool for C#

Sebastian Banescu
Technische Universität München
Boltzmannstr. 3
85748 Garching bei München, Germany
banescu@cs.tum.edu

Ciprian Lucaci
Technische Universität München
Boltzmannstr. 3
85748 Garching bei München, Germany
lucaci@cs.tum.edu

Benjamin Krämer
Technische Universität München
Boltzmannstr. 3
85748 Garching bei München, Germany
kraemerb@cs.tum.edu

Alexander Pretschner
Technische Universität München
Boltzmannstr. 3
85748 Garching bei München, Germany
pretschn@cs.tum.edu

ABSTRACT

Software protection is a difficult task especially for managed code, which executes only on a runtime environment such as C# or Java. Applications developed in such languages can be accurately decompiled, as opposed to x86 machine code. This facilitates reverse engineering attacks, with the goal of extracting proprietary algorithms. Due to the ease of distributing software copies across different jurisdictions, software developers cannot only rely on legal means for protection against reverse engineering attacks. Therefore, they have to employ technical means for software protection such as obfuscation. This paper presents an open source tool for virtualization obfuscation of programs written in the C# language, called VOT4CS. Our tool offers several possibilities for randomization that aim to confuse attacks based on pattern recognition. An evaluation of VOT4CS is performed based on several case-studies, which show the performance-security trade-off offered by the tool.

Keywords

Obfuscation, Software protection, Man-At-The-End Attacks

1. INTRODUCTION

The cost of developing commercial software leads some competitors to resort to intellectual property theft. If the software inside any product is not protected in any way, competitors can simply buy a copy of the product, reverse engineer the algorithms implemented in that product and use these algorithms inside their own product. Significant economic losses have been incurred by commercial software developers due to software cracking [2]. Proprietary software may be left unprotected by technical means, in the hope that legislation will be enough to deter attackers. However, software can be easily distributed to geographical areas where the laws regarding copyright are not aligned with laws where the software developer is located. As a result, commercial software developers often employ technical means of software protection to raise the bar against attackers who have physical access to the device on which their software runs.

One technical means of protecting intellectual property is called *software obfuscation*. Software obfuscation consists of code transformation techniques that aim to make computer programs harder to analyze, while preserving their functionality. In general, reverse-engineering obfuscated code cannot fully be avoided. The goal of obfuscation hence is to raise the bar for attackers to an extent where it is not worth investing the resources needed to deobfuscate it, relative to developing an equivalent software or buying a software license.

Virtualization obfuscation is a particular obfuscation technique which transforms the control-flow of a given software program by first translating the original code into another instruction set architecture (ISA) and then generating an interpreter for the corresponding ISA. Intuitively, such a transformation forces the attacker to first understand the semantics of the ISA before reverse engineering the underlying software. Virtualization obfuscation has been shown to be a strong obfuscation scheme in practice [2, 5, 22, 24]. However, it has been shown that virtualization does not stop reverse engineering attacks altogether [20].

Contributions: This paper presents the design and implementation of a virtualization obfuscation tool for C# programs called VOT4CS[1]. As opposed to existing tools such as such as Eazfuscator [14] and Agile.NET obfuscator [1], which operate on intermediate language, VOT4CS operates at the source code level. We are not aware of any free virtualization obfuscation tool for C# or .NET programs and therefore see a technical contribution in providing such a tool. Secondly, this paper performs a survey of the most important attacks against virtualization obfuscation and presents

[1]https://github.com/tum-i22/vot4cs

SPRO'16, October 28 2016, Vienna, Austria

ISBN 978-1-4503-4576-7/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2995306.2995312

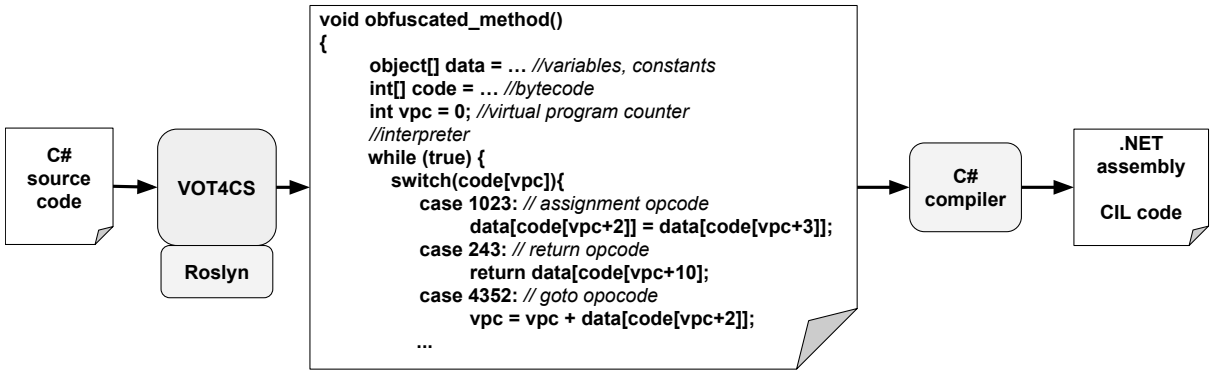

Figure 1: Virtualization obfuscation work-flow

an implementation of an attack based on this existing work. Thirdly, we perform a security and performance evaluation of VOT4CS based on a number of case-studies, which shows the trade-offs offered by our tool.

The remainder of this paper is organized as follows. Section 2 presents the design and implementation of VOT4CS. Section 3 presents reverse engineering attacks on obfuscated software with a focus on attacks against virtualization obfuscation. Section 4 presents the evaluation of our implementation. Section 5 presents related work in the area of software obfuscation. Finally, Section 6 discusses conclusions and future work.

2. DESIGN AND IMPLEMENTATION

There are two levels where one could perform obfuscation transformations of C# applications, i.e. intermediate language level and source code level. The option of applying the transformations after compilation at the *Common Intermediate Language* (CIL) level, is performed by commercial obfuscators (e.g. Eazfuscator [14], Agile.NET obfuscator [1]), because this makes the tool applicable to any programming language that is compiled to CIL (e.g. C#, Visual Basic .NET). VOT4CS uses the other option of applying the transformations before compilation, at the source code level. We chose this option, because manipulation of C# source code gives access to advanced features of the programming language (e.g. delegates), which can be included in the handlers of the virtualization interpreter and allows for compiler optimization, which will increase the performance of the obfuscated program.

The work-flow of virtualization, which involves VOT4CS is shown in Figure 1. For the implementation of VOT4CS we have used the .NET Compiler Platform ("Roslyn") [23], which enables easy manipulation of C# source code. Roslyn provides an abstract syntax tree view of the processed source code which facilitates transformation at source code level. Transformations applied by VOT4CS are grouped in two phases: *refactoring* and *virtualization* transformations, which are both described in the following subsections. The *refactoring* phase is applied first and has the role of bringing the code into a canonical form, which leads to a more compact and efficient implementation of the *virtualization* phase. For ease of readability and to motivate refactoring, we first present the virtualization phase; however, note that it is temporally applied after the refactoring phase.

2.1 Virtualization phase

VOT4CS facilitates virtualization obfuscation at the level of granularity of individual functions of the input software, i.e. the user can specify the functions that must be protected, instead of being forced to obfuscate the entire program, which would have a higher performance impact. The virtualization transformation is applied to the input program and it consists of the following three steps:

1. Map all variables, function parameters and constants in a function to an entry in a common `data` array, which represents the memory of the interpreter.

2. Map all statements in a function to a new randomly chosen language, which represents the *instruction set architecture* (ISA) of the interpreter. Store all the mapped statements inside a `code` array, which is called *bytecode* because it often consists of a sequence of bytes[2].

3. Create an interpreter for the previously generated ISA, which can execute the instructions in the `code` array using the `data` array as its memory. The input-output behavior of this execution must be the same as that of the original program.

In the following subsections, we describe each of the three steps in more detail.

2.1.1 Step 1: Map all data items to data array

All constants, local variables and method parameters are mapped onto an entry in a `data` array of type `object` since it is the most general data type in C#, which can accommodate any object or primitive type variable (see middle of Figure 1). If there are constants which are repeated in the original code, then they are mapped to the same entry in the `data` array. If there are local variables which are not initialized, then they are initialized with a random value. This makes the `data` array more resilient to tampering, because if the attacker modifies one value in the array targeting a specific statement that uses that value, s/he will probably also affect other statements in the original code, which share the same entry in the `data` array.

[2]In our implementation the `code` array is of integer type (size equal to a memory word), therefore the term *bytecode* is somewhat misleading. However, we use the term bytecode, because it is more familiar than “wordcode” or “intcode”.

2.1.2 Step 2: Map all instructions to code array

Each statement from the original program is mapped to an instruction in a randomly generated language (ISA). An instruction consists of an *operation code* (opcode), one or more operands and zero or more random values. Opcodes are randomly generated integer values each of which corresponds to a different type of statement from the original program. In the example from the middle of Figure 1: assignment statements are mapped to an instruction having the opcode 1023, return statements are mapped to an instruction with opcode 243 and goto statements are mapped to an instruction with opcode 4352. Each operand is an index of an element from the `data` array. All instructions are stored in a `code` array of integer values. To hamper static analysis (e.g. pattern analysis) of the new instructions in the `code` array, a random number of random integer values are interleaved between each opcode and operands, also between operands.

2.1.3 Step 3: Creating the interpreter

An interpreter for the random ISA generated in step 2, is created in step 3. The interpreter consists of an infinite `while` loop, which has a `switch` statement inside (see middle of Figure 1). Each `case` section of the `switch` statement is an opcode handler, i.e. each possible opcode in the bytecode program is processed by a dedicated part of the interpreter. The current instruction to be processed by the interpreter is indicated by an integer variable of the interpreter called the *virtual program counter* (VPC). The VPC is used to index the instructions in the `code` array and it is initialized with the offset of the first instruction in that array. In every instruction handler the operands of the current instruction are used to perform the operation(s) corresponding to this instruction. Afterwards, the VPC is set to the offset of the following bytecode instruction to be executed.

At the end of the virtualization phase, the body of an obfuscated method will be replaced by the `data` and `code` arrays, which have an associated interpreter.

2.2 Refactoring phase

Programming languages such as C# allow writing code having the same functionality in various ways. For instance, `for`-loops can be also re-written as `while`-loops having the same number of iterations. Another example is comparing if variable a is less than b, which can be done either by $a < b$ or $b > a$. This implies having a large variety of instruction handlers in the virtualization interpreter corresponding to different statements in the original code. For the previous examples we would have instruction handlers for: `for`-loops, `while`-loops, $<$ and $>$. However, this is neither beneficial for performance, nor for potency against tampering attacks, because the attacker may identify and disable an instruction handler corresponding to an integrity or license check faster than if multiple statements in the original code would share the same instruction handler in the interpreter. Therefore, in the refactoring phase, we bring the code into a canonical form before virtualizing it. The refactoring transformations presented next bring the code into a canonical form, without affecting the input-output behavior of the C# program given as input to VOT4CS.

2.2.1 Refactoring if-statements

The first refactoring transformation splits an `if` statement into two parts: (1) an assignment of the `if` condition to a boolean variable called `cond`, and (2) the `if` statement based only on the value of `cond`. Intuitively this transformation is similar to how a conditional statement appears in assembly language, i.e. first a compare instruction which sets the value of the flags register and then a conditional jump instruction based on the value of a certain flag. This transformation decouples the boolean condition from the `if` statement, which allows the virtualization interpreter to have a generic opcode handler for any possible `if` statement. The evaluation of the `if` condition and the assignment of its value to the variable `cond` will be performed by other opcode handlers. This means that an attacker cannot simply disable the opcode handler for `if` statements to disable a license check, because this will also break all other `if` statements in the program. To make matters worse for the attacker, we also transform `switch` statements into a series of nested `if` statements.

2.2.2 Refactoring loops

The second refactoring transformation transforms `for` and `do-while` statements into `while` statements. Additionally these `while` statements are transformed exactly like `if` statements, i.e. the condition is assigned to a boolean variable `cond` and the `while` statement is based only on `cond`. The intuition for this transformation is that `while` statements are similar to `if` statements, i.e. a boolean condition and a jump to a particular code location based on its value. Therefore, we will be able to use the same opcode handler for both `if` and `while` statements in the virtualization interpreter, which further exacerbates the task of reverse engineering the `code` array and tampering with the interpreter.

2.2.3 Refactoring unary and binary operators

The third refactoring transformation decreases the number of different unary and binary operators in the source code, because this leads to fewer opcodes in the random ISA. Fewer opcodes entails a more compact interpreter, which is more efficient from a performance point of view. Moreover, it also entails better protection against reverse engineering and tampering attacks as mentioned before. For example, multiple instructions which involve the same operation in the original program, will be mapped to the same opcode handler in the virtualized program. Therefore, tampering with the code of the interpreter by removing an opcode handler which is used for some sensitive action (e.g. authentication, authorization, etc.), will probably also affect other parts of the logic in the original program.

For this refactoring transformation we reduce the number of possible comparison operations from four to two. For example, `b>a` becomes `a<b` and `b>=a` becomes `a<=b`. For each assignment we transform the statement into its equivalent by using only simple operators. For example, `a+=b` becomes `a=a+b`. The same kind of transformation is also applied to composed, pre- and post-fix operators, i.e. `+=`, `*=`, `&=`, `|=`, `≪=`, `^=`, `%=`, `++var`, `var++`.

2.2.4 Refactoring statements with multiple operands

A statement can have multiple *operands*, e.g. `sum=a+b+c` has three operands, i.e. `a`, `b` and `c`. To reduce the number of opcode handlers in the interpreter even further, we refactored all statements such that they have the *minimum* number of operands allowed for a certain operation, i.e. 2 operands for binary operators and 1 operand for unary operators. To achieve this we create intermediate local variables each of

which are assigned values of sub-expressions with the minimum number of operands.

Our hypothesis at this refactoring step is that *fewer operands per statement entails fewer instruction handlers, which offers a better protection against reverse engineering attacks*. In order to test our hypothesis we allow the user of VOT4CS to specify the maximum number of operands that are allowed during refactoring. Therefore, if the user decides to set this input to ≥ 3 then, expressions such as `sum=a+b+c`, will be preserved in a dedicated opcode handler of the virtualization interpreter, which adds three operands. In Section 4 we present empirical results regarding the validity of this hypothesis.

2.2.5 Refactoring multiple invocations

Similarly to refactoring statements with multiple operands, we also refactor statements that access multiple class fields and statements which contain multiple method invocations. For example, the following statement `a.b().c().d();` has 3 method invocations. It can be split into 3 statements each containing one invocation. The result of each invocation is assigned to temporary variables, i.e. `x=a.b(); y=x.c(); y.d();`.

The hypothesis we use here is that *fewer invocations per statement entails fewer instruction handlers, which offers a better protection against reverse engineering attacks*. In order to test this hypothesis, VOT4CS also allows the user to specify the maximum number of invocations allowed per statement. We will present empirical results regarding the validity of this hypothesis in Section 4.

2.3 Raising the Bar for Attackers

The refactoring transformations presented in Section 2.2 are aimed at both reducing resource consumption by having a more compact interpreter and increasing the resilience of the virtualized program against reverse engineering and tampering attacks. In the following we present a few more features of VOT4CS which are strictly aimed at increasing resilience against attacks.

2.3.1 Software Diversity via Randomization

The idea of software diversity is to distribute syntactically different variants of the same program to different end-users, i.e. all variants have the same input-output behavior. The rationale behind this idea is to minimize the impact of an attack across a large set of installations of the same software, because attacks tailored for one software variant will not be effective against other variants [15].

The virtualization obfuscation mechanism described thus far features software diversity via the ISAs, which are randomly generated every time VOT4CS is executed. For example, the random opcodes of the virtualization interpreter are random integers (see middle of Figure 1). However, we have added other software diversity features into VOT4CS, by randomizing various aspects of the `code` array:

- The position of operands and the opcode within `code` array instructions is randomly chosen.
- The size of each bytecode instruction is randomly chosen.
- The most frequent opcode from the bytecode is assigned the opcode handler on the `default` section of `switch` statement of the interpreter. This is meant to raise the bar against frequency analysis attacks, because each occurrence of the most frequent opcode can be replaced by a random value (excluding the values of the other existing opcodes), in the `code` array.

These random choices allow generating a large number virtualized versions of the program which have the same input-output behavior. It also hampers pattern analysis in the `code` array and reuse of tampering attacks via patching scripts that modify the `code` array of different virtualized variants of the same program.

2.3.2 Interpreter level

The level of the interpreter can be set either at method or class level. The *method level interpreter* generates an interpreter which covers only the statements of one method. The *class level interpreter* generates a single interpreter which covers the statements from all obfuscated methods in that class, which means that statements form different methods share the same opcode handler of the interpreter, hence, tampering attacks are harder. Moreover, a class level interpreter makes reverse engineering and tampering attacks more difficult, due to the fact that not all opcode handlers may be used by all methods, e.g. an attacker who targets a particular method, wastes time analyzing more code (opcode handlers), which are not relevant for the targeted method.

3. ATTACKS

This section briefly describes the capabilities that so called *man-at-the-end* (MATE) attackers have at their disposal, in order to reverse engineer programs written in C#. It then presents existing work that aims specifically at reverse engineering virtualization obfuscation. Finally, it describes our implementation of an attack based on one of these existing approaches.

3.1 Man-At-The-End (MATE) Attacks

Anckaert et al. [2] introduced two categories of threats relevant for MATE attacks, i.e. *malicious code* and *malicious host*. *Malicious code* represents any kind of software (including viruses, worms, spyware, etc.), which is intended to execute potentially damaging actions for the end-user or the software developer (e.g. injecting unwanted adds into the web-browser). A *malicious host* is a machine owned by an attacker. In this case the attacker can directly access the software executables and perform any type of static and/or dynamic analysis on them. In this paper we will discuss both categories of MATE attacks.

C# applications are compiled into a .NET assembly, which is the piece of software shipped to end-users. Such assemblies consist of *Common Intermediate Language* (CIL) code, an object-oriented assembly language, which is entirely stack based. Using decompilers such as JustDecompile [18] and ILSpy [17], one can transform a .NET assembly into C# source code similar to the original program. The fact that .NET assemblies can be soundly and correctly decompiled back into C# code, makes a MATE attacker a likely threat for commercial software developers who distribute .NET assemblies to their end-users. This motivates the contribution of this paper, as VOT4CS would be used for the protection of software via virtualization obfuscation to raise the bar against MATE attackers.

Author	Attack Type	Automatic	Attacker Goal	Drawbacks
Rolles [22]	manual static analysis	partial	extract original code	time consuming, not scalable
Kinder [19]	automated static analysis (abstract interpretation)	yes	approximated data values	strong assumptions about interpreter structure
Sharif [24]	automated dynamic analysis and static analysis	yes	control flow graph	strong assumptions about interpreter structure
Coogan [10]	automated dynamic analysis (taint analysis)	yes	approximation of original code, significant trace	difficult to convert equations to control flow graphs
Yadegari [3]	automated dynamic analysis (bit-level taint analysis)	yes	control flow graph	large input space

Table 1: Deobfuscating virtualization obfuscation techniques

3.2 Attacks on Virtualization Obfuscation

Virtualization obfuscation poses challenges to existing de-obfuscation tools. According to Rolles [22] static analysis can only analyze the code of the random ISA interpreter, while the logic of the original program is hidden in the bytecode. If static analysis is to be more effective, the attacker needs to build a decompiler for the bytecode. Since a new ISA can be randomly generated each time VOT4CS runs, this decompiler must be generic. As far as the authors of this paper are aware, a fully-automated generic decompiler against virtualization obfuscation, has not been developed yet.

There has also been active research in developing dynamic analysis attacks or hybrid-analysis attacks to deobfuscate virtualization [3,4,10,19,24]. A summary of different deobfuscation approaches is listed in Table 1, which shows the author names (1st column), the type of attack (2nd column), if the attack can be automated (3rd column), the goal of the attack (4th column) and the drawbacks of the attack (5th column).

Sharif et al. [24] first locate the bytecode using dynamic analysis. Afterwards, they use static analysis to map the bytecode instructions to a specific handler of the interpreter. Finally, they recover the control-flow graph of the original program. However, this approach cannot properly handle ISAs with variable instruction lengths such as those generated by our implementation.

Kinder [19] uses abstract interpretation and develops an technique called *VPC lifting* in order to automatically discover internal variable values at any point when a certain function is called in the interpreter. Banescu et al. [4] employ dynamic symbolic execution to automatically generate valid inputs that allow bypassing a license checker obfuscated using virtualization. However, these two attacks do not return a simplified version of the algorithm used by the original program, which is the goal of the attacker motivated in Section 1.

The approaches of Coogan et al. [10] and Yadegari et al. [3] employ dynamic analysis to generate traces of the obfuscated program, which are simplified afterwards using multiple technique including taint analysis. The simplified traces are compared to traces of the original (un-obfuscated) program. The intuition being that the original algorithm of the program can be reconstructed given enough traces as demonstrated by Yadegari et al. [3]. They show that it is possible to construct control-flow graphs from the simplified traces and compare them to the control-flow graph of original program.

We believe trace comparison gives a similar "security score" as comparing control-flow graphs, while being more straightforward to perform. Therefore, in this work we decided to implement an attack similar to that of Coogan et al. [10], in order to test the resilience of VOT4CS against dynamic MATE attacks. Our implementation of this attack is presented in Section 3.3.

3.3 Attack Implementation

Since there is no available de-obfuscation tool against virtualization obfuscation for .NET applications at CIL level, we implemented one ourselves. We have implemented a dynamic attack based on the approach of Coogan et al. [10]. It consists of 2 steps, which are presented in the following subsections.

3.3.1 Tracing the program

In order to perform a dynamic trace analysis attack similar to the approach of Coogan et al. [10], we need to record execution traces of the obfuscated application. For applications written in C# we could record traces at two levels of abstraction, i.e. x86 assembly level or CIL level. Traces of x86 assembly instructions are more difficult to reverse engineer by an attacker than CIL traces, because the former are at a lower-level of abstraction. Therefore, even though Intel Pin [21] is an excellent x86 assembly level tracer, we argue that a skilled attacker would choose to build a tracing tool at CIL level in order to speed-up the analysis.

To trace code at CIL level we instrument any given C# program at CIL level, i.e. we insert instructions that record each executed CIL instruction of a C# program at run-time. As an instrumentation framework we use ConfuserEx [9] an open source protector for .NET applications at CIL level.

Directly comparing the trace of an un-obfuscated program with its virtualized counterpart is unfair, because the latter would generate larger traces than the former, due to the extra instructions of the virtualization interpreter. Moreover, this would assume that the attacker is unaware of the obfuscation transformation applied to the C# program. However, we assume that the attacker is aware of all the details of the VOT4CS tool, which practically eliminates any *security by obscurity*. This is a realistic attacker model if the obfuscation tool is open source, as is the case of VOT4CS. Therefore, we incorporate this knowledge of VOT4CS into our attack implementation, i.e. if the application is protected by virtualization obfuscation, we also log the value of the current opcode and the current VPC value.

		Original	Refactored-only					Refactored & Virtualized				
VOT4CS settings		none	op2 in1	op3 in1	op4 in1	op4 in4	op4 in5	op2 in1	op3 in1	op4 in1	op4 in4	op4 in5
Recorded trace	5	289	284	259	325	313	313	2498	2030	2092	1824	1687
	25	1149	1104	999	1285	1233	1233	9538	7450	7972	7224	6587
	125	5449	5204	4699	6085	5833	5833	44738	34550	37372	34224	31087
Simplified trace	5	280	275	250	316	304	304	295	234	331	307	289
	25	1120	1075	970	1256	1204	1204	1095	814	1231	1167	1069
	125	5320	5075	4570	5956	5704	5704	5095	3714	5731	5467	4969

Table 2: Recorded trace lengths for original, refactored-only and virtualized versions of the same program. The different columns under refactored-only and virtuazlied show the different VOT4CS settings used. The last two 3-tuples of rows for recorded trace and simplified trace indicate the different values for the program input, i.e. 5, 25 and 125.

3.3.2 Simplifying the trace

After the trace is generated, our attack implementation is followed by a simplification of the trace similar to that of Coogan et al. [10] and Yadegari et al. [3]. Our trace simplification is meant to remove the instructions added by virtualization obfuscation and only keep the instructions which represent the functionality of the original (unobfuscated) program, hence making the trace easier to understand by an attacker. Therefore, we have implemented the following trace simplification steps:

1. Filter out all of the intermediate instructions which belong to `switch` or `if-else-if` statements.
2. Filter out instructions which increment the VPC.
3. Replace the sets of instructions accessing the `data` array based on the value of `code` array indexed by VPC with an instruction indicating a variable access.

The index in the `data` array is appended to the name of the variable, instead of giving it a random name. This way the attacker knows when a variable is reused in different sets of instructions of the trace. For example, after trace simplification it is easier to spot a loop iterator variable from the original program, because it is incremented several times at fixed intervals.

4. EVALUATION

First of all an obfuscation tool that performs virtualization obfuscation should preserve the input-output behavior of the input program. Proving behavior equivalence is a difficult problem, which we did not tackle in this work. Therefore, during the development of VOT4CS we have made use of the concept of *observational equivalence* [7, 10] in the following way. For each obfuscation sample, for the same input we compared the output result with the output of the original sample. For each method we run the comparison with multiple sets of inputs such that we have complete statement coverage of the original program.

If the outputs of the obfuscation tool are correct, then the rest of the evaluation is generally performed according to the four criteria of Collberg's taxonomy [8], i.e.: (1) *potency* refers to the difficulty of the obfuscation to be understood by humans, (2) *resilience* refers to the difficulty of the obfuscation to be bypassed automatically by a de-obfuscation technique, (3) *cost* refers to the computational overhead added by the obfuscation and (4) *stealth* refers to the detectability of the obfuscated code in the rest of the code.

Since potency requires an experiment involving human subjects, which is laborious to setup and lengthy to describe, we leave it for future work. Some existing user studies already show that weaker obfuscation techniques than virtualization are quite potent [6]. Another study by Rolles [22] argues that using a debugger to reverse engineer virtualization obfuscation is tedious because one must repeatedly inspect the interpreter parsing code, i.e. there is a low signal to noise ratio. Therefore, the high-level details are obscured by the flood of low-level analysis.

Due to the typical code footprint of the interpreter, virtualization obfuscation can be easily detected and therefore its stealth is relatively low compared to most obfuscation techniques. Therefore, we do not believe this criteria requires further investigation for the particular implementation presented in this paper. In this work we mainly focus on the remaining criteria from Collberg's taxonomy, i.e. resilience and cost, because they allow us to present concrete and objective measurements.

4.1 Resilience

To measure the resilience of VOT4CS against the attack described in Section 3.3, we constructed a function (see Appendix), which takes one integer as an input parameter and processes data using multiple arithmetic operations and functions calls. This function was written such that all the refactoring transformations presented in Section 2.2, could be applied and multiple statements in the program would share instruction handlers in the virtualization interpreter. We picked a set of inputs consisting of integers 5, 25 and 125, to check the increase in trace size as a function of the input value of the function. Since the refactoring transformations may also introduce differences between the original and the obfuscated versions of the same program, we have also generated another variant of the original program where we applied only the refactoring transformations (not the obfuscation transformations). On top of the refactored-only program we applied the virtualization obfuscation transformation described in Section 2.1.

We record traces at CIL level for the original, refactored-only and obfuscated programs using the tracer we presented in Section 3.3.1. We simplify the trace of the obfuscated program using the algorithm described in Section 3.3.2. After trace simplification the length of the traces from the virtualized programs are reduced to a size close to the trace of the original program, as shown in the bottom-half of Table 2. The first row of the table shows the settings we used for

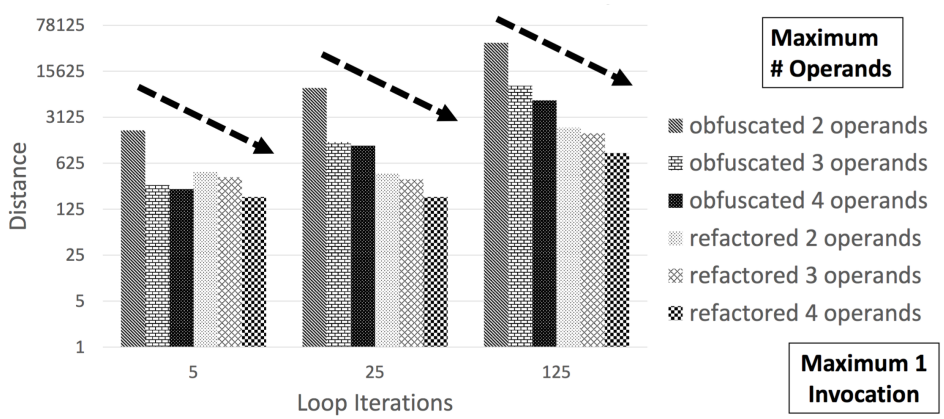

Figure 2: Trace setting - maximum number of operands

VOT4CS, i.e. the number of operands denoted as "op" (see Section2.2.4) and the number of invocations denoted as "in" (see Section2.2.5). The second row of the table is a 3-tuple that shows the length of the recorded traces before simplification for each of the three program inputs, i.e. 5, 25 and 125. The third and final row (3-tuple) of the table shows the length of the recorded traces after simplification.

Note that this simplification may also remove instructions which are part of the original logic of the program, if the original program contains patterns that are matched by our previously mentioned trace simplification algorithm. This results in simplified traces whose length is shorter than that of the original program. Therefore, the attacker will lose some semantics of the original program and therefore will have to infer these missing statements.

Since the simplified traces may: (1) still contain some instructions added by obfuscation and (2) miss instructions removed by the simplification algorithm, we argue that a valid metric for comparing a simplified trace of an obfuscated program with a trace of an original program is the *Levenshtein distance*, also used in plagiarism detection [26]. This measure intuitively gives the minimum number of operations that the attacker would have to make to recover the original trace, which is needed to reach the algorithm in the original program. The higher the value of the Levenshtein distance, the less similar the two compared traces are. We assume that a lower similarity means that the attacker must invest a higher effort to understand the original algorithm in the unobfuscated program.

However, a bitwise comparison of the two traces would not be accurate because it takes into account several syntactical constructs which are irrelevant for the control-flow, e.g. extra spaces, different variable names. Therefore, we abstract all traces according to the following criteria:

1. Loading a variable, argument or constant are considered the same.
2. Storing a variable, argument or constant are considered the same.
3. Only function names are compared, their parameters are discarded.

The intuition behind the the first and second criteria is that in virtualized programs, the arguments and constants of a functions are put in the `data` array which is treated as a variable. The intuition behind the third criteria is that the refactoring steps of VOT4CS will limit the number of arguments of function calls and this will differ w.r.t. the original program.

4.1.1 Number of operands

Here we show results that confirm our hypothesis from Section 2.2.4, i.e. *fewer operands per statement entails fewer instruction handlers, which offers a better protection against reverse engineering attacks*. We computed the Levenshtein distance between the simplified trace of the original program and each of the simplified traces generated by VOT4CS with the number of invocations set to 1 and the number of operands was varied over values 2, 3 and 4. We do not show a comparison for a larger number of operands than 4 because it is practically the same as having the number of operands equal to 4 for the program we have tested. The results are shown in a bar plot in Figure 2 and they are the same traces indicated in the bottom half of Table 2, under the corresponding settings. Note that a lower value for the VOT4CS setting indicating the number of operands, will result in a more secure program, because of the higher Levenshtein distance w.r.t. the trace generated by the original program.

4.1.2 Number of invocations

Similarly, we show results that confirm our hypothesis from Section 2.2.5, i.e. *fewer invocations per statement entails fewer instruction handlers, which offers a better protection against reverse engineering attacks*. We computed the Levenshtein distance between the simplified trace of the original program and each of the simplified traces generated by VOT4CS with the number of operands set to 4 and the number of invocations was varied over values from 1 to 5. Figure 3 shows that reducing the number of invocations to 1 in the settings of VOT4CS, will produce the most secure program, because its Levenshtein distance will be the highest w.r.t. the trace generated by the original program. Note that having a larger number of invocations than 3 results in a small change in the Levenshtein distance, because there is only one statement which contains more than 3 chained invocations.

4.2 Cost

Run-time overhead: To measure the run-time overhead added by virtualization obfuscation to a program, we picked two types of programs: (1) single function programs that are CPU intensive (e.g. sorting algorithms and searching algo-

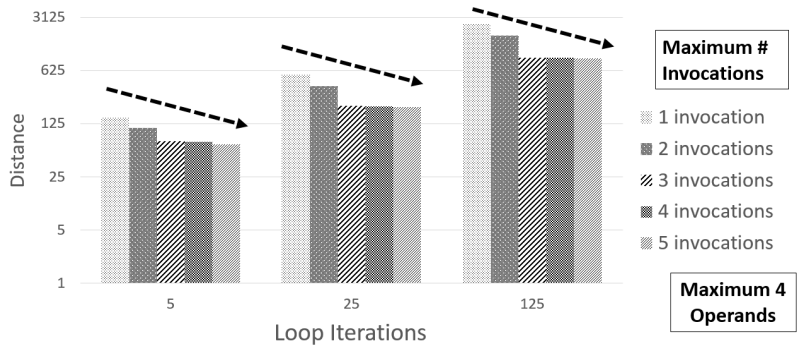

Figure 3: Trace setting - maximum number of invocations

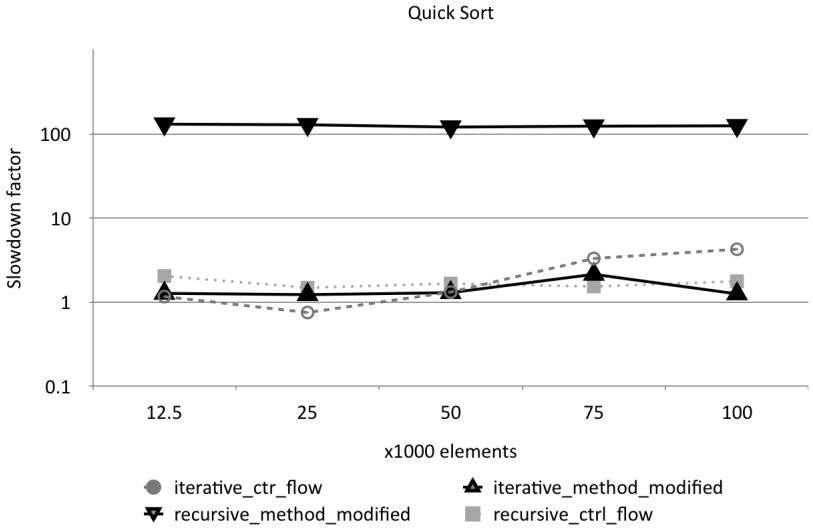

Figure 4: QuickSort - iterative and recursive virtualization and control-flow

rithms), (2) larger projects consisting of hundreds of methods, out of which we only obfuscate a subset of the methods. The first category gives us the worst-case scenario. The second category is similar to the use of virtualization in real-world applications, where a software developer would only obfuscate the most important parts of an application (e.g. a license checking mechanism, a secret algorithm that brings an advantage w.r.t. competitors, etc.).

To simulate the worst-case scenario for run-time overhead, we set the following input parameters of VOT4CS: *maximum number of operands* set to 2, *maximum number of invocations* set to 1. We picked *quick-sort* and *binary-search* as single function programs that are CPU intensive. We recorded 125 runs of the quick-sort algorithm on 5 different numbers of input elements, i.e. inputs with 12.5K, 25K, 50K, 75K and 100K integer elements. The slowdown factor varied between 2 and 4, which we believe is acceptable for some applications. We also executed 125 runs of the binary-search algorithm with inputs of 500K, 1M, 2M, 3M, 4M integer elements and the slowdown factor varied between 38 and 84, which is also deemed acceptable, but in fewer application scenarios.

With regard to the performance of single function programs we compared VOT4CS with ConfuserEx's implementation of control-flow obfuscation. In Figure 4 we observe that for the iterative version of quick-sort, control-flow obfuscation performs better than virtualization for fewer elements, but with an increasing input size the performance of control-flow is lower than virtualization. However, for a recursive implementation of quick-sort control-flow obfuscation performs better than virtualization in every circumstance. Due to the fact that quick-sort is an allocation and deallocation intensive operation virtualization introduces a major slowdown due to variables allocation in the `data` array.

In the case of the binary-search algorithm, see Figure 5, which performs a high number an array accesses, for the iterative implementation of the algorithm, virtualization obfuscation performs better than control-flow obfuscation. ConfuserEx's implementation of control-flow obfuscation uses labels for implementing jumping and mathematical operations. Therefore, if control-flow is optimized with more efficient operations it outperforms virtualization. However, control-flow obfuscation does not reuse code for operations and there is no variable sharing such as offered by virtualization.

On the other hand, even in one medium sized project there are thousands of methods and one use-case could potentially trigger hundreds of method invocations. Therefore, we are interested in how the overall project performance is affected by virtualizing only a few methods of the project. For this purpose we used the source code of an open source project

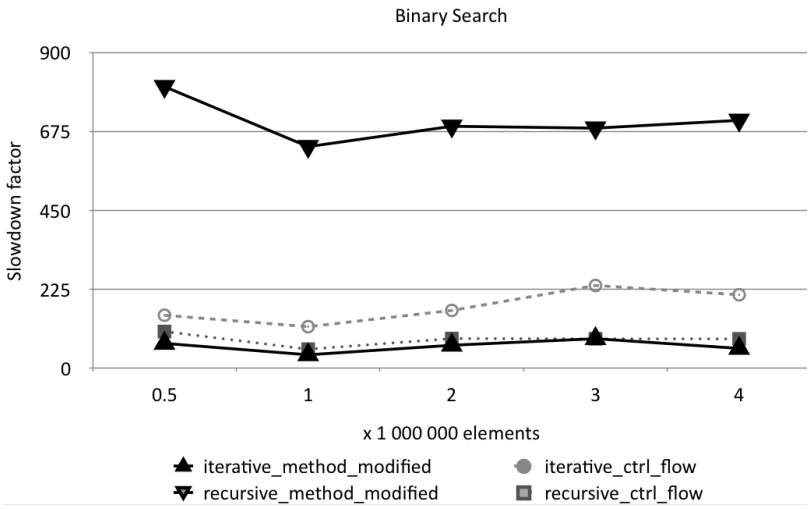

Figure 5: Binary Search - iterative and recursive virtualization and control-flow

	Original	Obfuscated	Relative Increase
Quick Sort	479,232	491,520	2.57%
Binary Search	483,328	491,520	1.70%
ResourceLib	75,264	145,920	93.87%

Table 3: Virtualization obfuscation size impact in bytes

called *ResourceLib*[3], a file resource management library. The project consists of more than 100 methods spread over more than 40 classes. Out of these we have obfuscated 8 methods in 4 different classes. The methods were selected based on the complexity of their computations and the code constructs that our tool supports. They range from simple methods with branching structures to more complex methods with multiple loops. The interpreter was at the method level and the refactoring settings were set to most aggressive. We have timed the total runtime of 46 unit tests. During the automated run of the unit tests there have been more that 1300 calls to the obfuscated methods. Each total runtime of the 46 unit test was measured with .dotTrace [13]. We have performed the measurements 25 times and the results show only a 0.5 slowdown w.r.t. the original program.

File size overhead: There is an impact on the .NET assembly size of the obfuscated programs, because every extra line of code added in the process requires extra space. We have measured the file size of the managed code assembly on the disk. Quick Sort and Binary Search methods were measured in an assembly with additional modules. We measured the same method in the original form and in the obfuscated form. For ResourceLib we measured the entire compiled project. The results vary between a 1.70% to a 93.87% increase in size and are listed in Table 3. ResourceLib has a small disk footprint therefore adding a few hundreds lines of code can lead to significant size increase, whereas in the case of a larger assembly having only a few methods obfuscated has no significant implact on the overall size.

Summary: The results of runtime and size analysis show an increase in both execution time and storage space. However, if virtualization is not applied to the whole program but only to some functions, as expected for real applications, the size and runtime overhead is acceptable for many application scenarios.

[3]https://github.com/dblock/resourcelib

4.3 Limitations

VOT4CS does not virtualize lambda expressions, because Roslyn does not support all features of the C# language, e.g. field reassignment [16]. VOT4CS does not support *try/catch* statements, because it is not clear how to implement the *finally* clause as an instruction handler. VOT4CS cannot store the class attributes in the `data` array without affecting the visibility of the field and its use in other methods. A compiler limitation exists with regard to method parameters which have the type `ref` and `out`, because the compiler does not allow referencing a `cast` construct from the element of an array [25]. Another compiler limitation occurs with `struct` types, i.e. after storing the `struct`'s identifier to the `data` array, the compiler will give errors when trying to use casting when accessing the attribute of that `struct` element.

5. RELATED WORK

There are a couple of ways to protect a software against malicious host attacks. Collberg's taxonomy [7] of software protection techniques is very helpful in understanding these available options.

Legal: The easiest but also least efficient option would be to make use of intellectual property laws and to enforce them by bringing any offender to court. Unfortunately, economic realities often make it difficult to enforce the law. Also having clients from different countries, inconsistencies between legal systems, makes protecting intellectual property quite challenging.

Remote services: One of the most secure approaches with regard to reverse engineering would be to not sell application copies, but rather sell it as a remote service. In this way, no user will ever gain access to the application itself. By never gaining direct access to the application binary itself, the attacker will not be able to reverse engineer it. But this approach has a few obvious limitations such as network bandwidth or running the application in an environment where

Category	commercial					non-commercial	
Tool name	Agile.NET	Crypto Obfuscator	Eazfuscator	Dotfuscator	AxProtector	Tigress	ConfuserEx
Vendor	[1]	[11]	[14]	[12]	[27]	[28]	[9]
Price	$795	$399 - $4.469	$399	ca. $1.600	$100	**Free**	**Free**
Languages	C#	C#	C#	C#	C(++), C#, Java	C(++)	C#
Code Virtualization	**Yes**	**Yes**	**Yes**	-	-	**Yes**	-
Diversifying	**Yes**	-	-	-	**Yes**	**Yes**	-
Symbol Renaming	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	-	**Yes**
Control-Flow Obfuscation	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Watermarking	**Yes**	**Yes**	-	**Yes**	-	-	-
String Encryption	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	-	**Yes**
Code Encryption	**Yes**	**Yes**	-	-	**Yes**	-	**Yes**
Resource Encryption	**Yes**	**Yes**	**Yes**	-	**Yes**	-	**Yes**
Constant Encryption	-	**Yes**	-	-	**Yes**	-	**Yes**
Debug Detection	-	**Yes**	-	-	**Yes**	-	**Yes**

Table 4: Comparison of obfuscation tools

no Internet access is available (e.g. while traveling).

Encryption: Encryption would be another alternative to protect the code. The most secure way for this would be to have the entire decryption/encryption process in hardware but even hardware solutions are known to have limitations [2]. If the code is executed in a software virtual machine interpreter, such is the case for Java bytecode and C# managed binaries, then it will always be possible for an attacker to intercept and decompile the decrypted code.

Machine code: Software could be delivered as binary files specific for each client's architecture. Having access to only machine code makes the task more difficult but it raises code portability issues.

Obfuscation and others: To defend code against a malicious host, other techniques have been discussed at length in research literature, techniques including watermarking, obfuscation and tamper-resistance [2]. In this work we focused on obfuscation.

Tool comparison: There exist several tools for protecting software against MATE attacks. The tools differ in their functionality, the programming languages they can be applied to and their price. A comparison of a set of popular software protection tools is presented in Table 4, which contains the tool names on each column and the tool vendor, price, target languages and features on the rows.

There are a few commercial obfuscators for C# that offer code virtualization, such as Agile.NET [1], Eazfuscator [14] and Crypto Obfuscator [11]. However, their prices range from 399 USD for single developer licenses to 1000s of USD for multi-user licenses. Tigress is the only non-commercial tool that offers virtualization obfuscation, however, it only supports C programs. We believe VOT4CS is a good compliment for ConfuserEx [9], because ConfuserEx offers many other transformations that should be applied after virtualization obfuscation. Of course, VOT4CS can also be used in combination with commercial tools like AxProtector [27], which do not offer virtualization obfuscation.

6. CONCLUSIONS

This paper presents VOT4CS a virtualization obfuscation tool for C# programs. The purpose of VOT4CS is to raise the bar against reverse engineering and tampering attacks on C# programs, which are built as .NET assemblies. Due to the fact that .NET assemblies can be unambiguously decompiled back into C# source code, such attacks are critical.

We presented a survey of attacks on virtualization obfuscation and implemented a dynamic trace simplification attack based on pattern matching. Furthermore, we performed an evaluation based on several case studies with resource intensive programs and real-world applications. In our case-studies, we showed that programs protected by VOT4CS have a higher resilience against MATE attacks when refactoring transformations are performed such that the number of opcode handlers is minimized. Regarding performance we show that the impact on recursive versions of CPU intensive applications is high, however, it is low for less CPU intensive, real-world applications. Since the overhead induced by VOT4CS is acceptable for several application scenarios, we believe it is and important tool for defending against reverse engineering and tampering attacks against C# applications.

Directions for future work include performance improvements for the code generated by VOT4CS. Currently, only basic constructs of the C# language are supported, due to the Roslyn version available when the tool was developed. We aim to include more features of the C# language to VOT4CS, in future work, including handling multi-threaded functions, which are not included at the moment. Another challenge we are interested in tackling in future work is automatic equivalence checking of the input and output C# programs.

7. REFERENCES

[1] Agile.NET. Code Protection. http://secureteam.net/obfuscator.aspx. [Online; accessed 16-March-2015].

[2] B. Anckaert, M. Jakubowski, and R. Venkatesan. Proteus: Virtualization for Diversified Tamper-resistance. In *Proceedings of the ACM Workshop on Digital Rights Management*, DRM '06, pages 47–58, New York, NY, USA, 2006. ACM.

[3] B. W. B. Yadegari, B. Johannesmeyer and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691, 2015.

[4] S. Banescu, M. Ochoa, and A. Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *Proceedings of the 1st International Workshop on Software Protection*, SPRO '15, pages 45–51, Piscataway, NJ, USA, 2015. IEEE Press.

[5] J. Cazalas, J. T. McDonald, T. R. Andel, and N. Stakhanova. Probing the limits of virtualized

software protection. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, PPREW-4, pages 5:1–5:11, New York, NY, USA, 2014. ACM.

[6] M. Ceccato, M. D. Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074, Feb. 2013.

[7] C. Collberg, I. Clark, and T. D. Low. D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In *In: Proc. of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, 1998.

[8] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations, 1997.

[9] ConfuserEx. free, open-source protector for .NET applications. https://yck1509.github.io/ConfuserEx//. [Online; accessed 20-May-2015].

[10] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 275–284, New York, NY, USA, 2011. ACM.

[11] Crypto. Obfuscator For .Net. http://www.ssware.com/cryptoobfuscator/obfuscator-net.htm. [Online; accessed 20-May-2015].

[12] Dotfuscator. .NET Obfuscation. https://www.preemptive.com/products/dotfuscator/overview. [Online; accessed 20-May-2015].

[13] dotTrace. .NET Profiler. https://www.jetbrains.com/profiler/index.html. [Online; accessed 20-May-2015].

[14] Eazfuscator.NET. obfuscator and optimizer for .NET. http://www.gapotchenko.com/eazfuscator.net. [Online; accessed 16-March-2015].

[15] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 67–72. IEEE, 1997.

[16] M. Gravell. Expression as a compiler. http://www.infoq.com/articles/expression-compiler. [Online; accessed 12-July-2015].

[17] ILSpy. open-source .NET assembly browser and decompiler. http://ilspy.net/. [Online; accessed 20-May-2015].

[18] JustDecompile. .NET assembly browser and decompiler. http://www.telerik.com/products/decompiler.aspx. [Online; accessed 20-May-2015].

[19] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 61–70, Washington, DC, USA, 2012. IEEE Computer Society.

[20] T. László and A. Kiss. Obfuscating c++ programs via control flow flattening. In *Annales Univ. Sci. Budapest., Sect. Comp. 30*, pages 3–19, 2009.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[22] R. Rolles. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[23] Roslyn. The .NET Compiler Platform provides open-source C# and Visual Basic compilers with rich code analysis APIs. https://github.com/dotnet/roslyn. [Online; accessed 16-March-2015].

[24] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 94–109, Washington, DC, USA, 2009. IEEE Computer Society.

[25] stackoverflow.com. Passing an explicit cast as a ref parameter (c#). http://stackoverflow.com/questions/2165892/passing-an-explicit-cast-as-a-ref-parameter-c. [Online; accessed 12-July-2015].

[26] Z. Su, B.-R. Ahn, K.-Y. Eom, M.-K. Kang, J.-P. Kim, and M.-K. Kim. Plagiarism detection using the levenshtein distance and smith-waterman algorithm. In *Innovative Computing Information and Control, 2008. ICICIC '08. 3rd International Conference on*, pages 569–569, June 2008.

[27] W. Systems. Protection Suite - AxProtector - Automatic Protection. http://www.wibu.com/axprotector.html. [Online; accessed 05-December-2015].

[28] Tigress. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu. [Online; accessed 05-December-2015].

APPENDIX

Source code of function used for the experiments presented in Section 4.1.

Listing 1: Test function

```
private string f(int b) {
  string sum = "" + 3 + 4 + "";
  sum += car.GetEngine()
            .GetPiston(car.GetEngine().GetPistons().Count
                - 1)
            .ToString();
  string r = "";
  string[] dst = new string[b];
  for (int i = 0; i < ReturnArg_Array(b); i++){
    sum += "_" + i + "_";
    sum += "~";
    r += sum + "#";
    var p1 =
        car.GetEngine().GetPistons().First().GetSize();
    r += "[" + p1 + "]";
    sum += r.Length;
    dst[i] = sum;
  }
  sum += "#" + dst.Length;
  return sum;
}
```