Code Obfuscation Against Symbolic Execution Attacks

Sebastian Banescu
Technische Universität München
banescu@in.tum.de

Christian Collberg
University of Arizona
collberg@gmail.com

Vijay Ganesh
University of Waterloo
vganesh@uwaterloo.ca

Zack Newsham
University of Waterloo
znewsham@uwaterloo.ca

Alexander Pretschner
Technische Universität München
pretschn@in.tum.de

ABSTRACT

Code obfuscation is widely used by software developers to protect intellectual property, and malware writers to hamper program analysis. However, there seems to be little work on systematic evaluations of effectiveness of obfuscation techniques against automated program analysis. The result is that we have no methodical way of knowing what kinds of automated analyses an obfuscation method can withstand.

This paper addresses the problem of characterizing the resilience of code obfuscation transformations against automated symbolic execution attacks, complementing existing works that measure the potency of obfuscation transformations against human-assisted attacks through user studies. We evaluated our approach over 5000 different C programs, which have each been obfuscated using existing implementations of obfuscation transformations. The results show that many existing obfuscation transformations, such as virtualization, stand little chance of withstanding symbolic-execution based deobfuscation. A crucial and perhaps surprising observation we make is that symbolic-execution based deobfuscators can easily deobfuscate transformations that preserve program semantics. On the other hand, we present new obfuscation transformations that change program behavior in subtle yet acceptable ways, and show that they can render symbolic-execution based deobfuscation analysis ineffective in practice.

1. INTRODUCTION

This paper addresses the problem of characterizing the resilience of practical code obfuscation transformations against symbolic execution attacks, complementing existing work that measures the potency of obfuscation transformations against human-assisted attacks [15]. We consider widely-used obfuscation techniques (e.g., virtualization) and show them to be surprisingly ineffective against symbolic execution based analysis and deobfuscation. We propose new obfuscation techniques that change program behavior in subtle yet acceptable ways, and we show that they are effective against symbolic execution methods.

Obfuscators (a.k.a. obfuscation compilers or transformations, first formally defined by Barak et al. [8]), are programs that take as input arbitrary programs[1], and transform them such that the resulting code satisfies three properties: (1) it must be semantically equivalent to the corresponding input (*functionality property*), (2) be at most polynomially bigger or slower than the input program (*slowdown property*), and (3) be as "hard to analyze and deobfuscate" as a black-box version of the program (*virtual black-box property*). The functionality and slowdown properties can be stated with relative ease, but in general are difficult to prove and can be as hard as proving correctness of arbitrary compiler transformations. While we acknowledge the relevance of these concerns, the subject of this paper is the third property.

Dozens of different software obfuscation techniques have been published [19]. They are widely used by both malware developers and software companies to thwart static or dynamic analysis deobfuscation attacks. The sheer volume of different applications and malware variants that can be generated by obfuscators renders the task of human-assisted analysis unscalable. Therefore, attackers resort to automated analyses.

A central issue with today's practical obfuscating compilers is that they do not clearly define in how far they are resilient against automated or manual attacks. How can a user of an obfuscating compiler know whether the obfuscations advertised by the compiler writer can indeed withstand highly sophisticated static, dynamic, symbolic, or machine learning deobfuscation attacks? What kind of assumptions can one make about the resources available to the attacker? Can we hope for a happy-medium where an obfuscation transformation is as resilient as the cryptographic varieties and yet is practical enough that the slowdown is acceptable?

While we do not provide a general answer to the above questions, we provide perhaps the first systematic approach to studying obfuscation resilience against automated attacks based on symbolic execution. As far as the authors are aware, there is no standard methodology for characterizing the resilience of different obfuscation transformations w.r.t. each other, against automated attacks. We are aware of works that focus on the empirical evaluation of the po-

ACSAC '16, December 05 - 09, 2016, Los Angeles, CA, USA

ISBN 978-1-4503-4771-6/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2991079.2991114

[1]Without loss of generality, we assume that obfuscators are source-code transformers that take C programs as inputs.

tency of obfuscation against human-assisted analysis [14, 15]. However, according to the pioneering work of Collberg et al. [20], obfuscation strength should be measured in terms of both *potency* against human-assisted attacks and *resilience* against automated attacks. This paper is concerned with an empirical evaluation of the latter, with a focus on symbolic execution. Our work is complementary to existing work focused on human-assisted attacks, as well as works that evaluate the resilience of obfuscation against static analysis attacks [38, 24]. While focusing on symbolic execution may seem narrow in scope, it has been reported that a large number of deobfuscation techniques rely on symbolic execution [47].

We emphasize that our paper is empirical in nature, in contrast to analytical studies on cryptographic obfuscation [8, 29]. An analytical study about characterizing the resilience of different obfuscation transformations is difficult to perform because it would need to cater to *all* programs, possibly restricted to a family of programs. In this empirical study we apply obfuscation to limited datasets of programs. We make the datasets heterogeneous by varying a set of program characteristics that make us believe that our findings achieve some degree of generality.

This paper makes the following **contributions**:

1. Proposes an approach for empirically characterizing the resilience of code obfuscation transformations, based on the relative increase in slowdown of symbolic execution engines on obfuscated applications w.r.t. their unobfuscated counterparts (§ 2).

2. A case-study on over 5000 C programs, two obfuscation engines and two symbolic execution engines (§ 3). The results of the case study show the resilience of different obfuscation transformations w.r.t. each other against symbolic execution based attacks.

3. Proposes a way of improving existing obfuscation transformations against symbolic execution by altering the *functionality property* from the previously mentioned obfuscator definition (§ 4).

2. AUTOMATED ANALYSIS ATTACKS

Automated attacks do not involve any human interaction during their execution. Step-by-step debugging is one example of a human-assisted attack, which involves a high degree of human interaction. Effectiveness of obfuscation against human-assisted attacks is difficult to quantify since it not only depends on the features of the debugger, but also on the knowledge of the human operating it. Given the same deterministic obfuscated program and the same analysis tools (e.g. IDA Pro [25]), for any two different individuals, it will likely result in significantly different times to complete the analysis. On the other hand, executing an automated attack multiple times (with the same seed, if the attack is randomized), on the same obfuscated program results in the same analysis time minus a negligible delta due to other background processes and the OS scheduler. Automatic analysis attacks are relevant for malware analysis, where the number of different malware variants observed in the wild are in the order of millions per day [40, 52] and cannot be easily analyzed with human assistance. Generally speaking, automatic analysis attacks are relevant for scenarios where a software developer employs software diversity [26, 27] via obfuscation, i.e. different end-users run differently obfuscated versions of the same program and an attacker targets all obfuscated versions of that program. This means that the amount of computing resources an attacker can spend on an automated analysis attack must be limited in order to cope with the large amount of obfuscated versions. Moreover, automated attacks that target a large population of end-users often come in the form of *potentially unwanted programs* (PUPs) [40] or *changeware* [7], executed on end-user machines, i.e. commodity hardware on average.

2.1 A Common Subgoal of Automated Attacks

Schrittwieser et al. [46] note that motivations of reverse engineers are diverse. However, their goals can be placed under the following 2 categories: (1) extracting a proprietary algorithm or data (e.g. secret keys, credentials) from a program and (2) modification of software to change its behavior, also known as *software-tampering*. The goals in the first category can be achieved by simplifying the control-flow and data-flow of the target program such that all irrelevant constructs and instructions added by obfuscation transformations are discarded and only the parts essential for functionality are maintained [58]. The goals in the second category can be achieved by first identifying and disabling any integrity checks on the code itself [43] and then modifying the actual functionality of the target program. We believe that there is a common subgoal for both of these goals.

In order to identify this common subgoal we look towards state of the art automatic attacks. Such automatic attacks are often specific to certain implementations of obfuscation transformations (e.g. virtualization obfuscation [38, 35, 48, 22], opaque predicates [24], control-flow-flattening [53]) or certain attacker goals (e.g. control-flow graph (CFG) simplification [58], identifying code self-checks [43], bypassing license checks [6]) or both. Moreover, all of these attacks, except for those based on abstract interpretation [38, 24] use dynamic analysis often in combination with some form of static analysis. The main reason for the use of dynamic analysis is that obfuscation techniques such as run-time unpacking and self-modifying code cannot be analyzed statically. However, a pre-requisite of dynamic analysis is the generation of valid inputs for the program being analyzed. How these valid inputs are obtained is not always described in works that present automatic analysis attacks. In some works they are picked randomly, in others they are generated using a symbolic execution engine.

Random test case generation is not sufficient for common attacker goals. We argue that certain automated analysis attacks require generation of test suites that achieve up to 100% (reachable) code coverage. Firstly, consider the goal of simplifying the CFG of an obfuscated program as presented in [58]. In order to ensure that the CFG is complete (i.e. there are no missing statements, basic blocks or arcs), the analysis technique requires execution traces that cover all the code of the program being analyzed. Secondly, consider the goal of identifying code which verifies checksums of other parts of the code as presented in [43]. To ensure that all self-checking code instructions are identified (which is mandatory in case there is a cyclic dependency between the instructions which perform checking [16, 36]), the analysis technique requires execution traces that cover all the code of the program being analyzed. Thirdly, consider the

goal of bypassing a license check as presented in [6]. If the license check is performed by a conditional statement based on an input to the program, generating a test suite that covers all the code of the program guarantees that one of the test cases contains the license key. However, unlike the previous two goals, a test suite that achieves 100% code coverage is sufficient, but not necessary, because the license key may be guessed correctly before the test suite covers 100% of the code. Nevertheless, we believe that test case generation is a common pre-requisite for all state of the art automatic analysis attacks. Hence, our proposal in this paper is to characterize the resilience of obfuscation transformations based on the increase in the effort needed to generate test cases for the obfuscated program relative to its unobfuscated counterpart.

We acknowledge the fact that after achieving this subgoal, an automatic analysis attack may need to perform further tasks to achieve its end goal. However, those tasks are different for different attacker goals, while the subgoal of automated test case generation is common for most state of the art attacks. We claim that the effectiveness of an obfuscation transformation can be measured by the increase in effort (i.e. slowdown) for automated test case generation.

2.2 Automated Test Case Generation

The main techniques for automated test case generation according to Anand et al. [1] are: symbolic/concolic execution, model-based test case generation, combinatorial testing, fuzzing, (adaptive) random testing and search-based testing. By "automated" we mean that the user does not need to provide any specification to the test generation method, only the program source or binary code is needed. These test case generation techniques can be further divided into *white-box testing* techniques (i.e. symbolic/concolic execution), which analyze the program code to guide test case generation and *black-box testing* techniques (i.e. model-based testing, combinatorial testing, fuzzing, adaptive random testing and search based testing), which do not analyze but simply run the code.

Since obfuscation techniques change the code of a program but not its input-output behavior, they only affect white-box testing techniques, modulo any overhead which is also incurred by black-box testing techniques due to executing any additional instructions in the obfuscated version of a program. We hence argue that the effectiveness of applying one or more obfuscation transformations can be measured by the increase in effort needed for a white-box testing technique to generate a test suite that covers all the code of the given program (e.g. for the goal of CFG simplification) or to find an input that leads to a certain execution path (e.g. guessing a license key).

If the absolute effort for a white-box testing technique to generate a test suite for an obfuscated program surpasses the effort for a black-box testing technique to generate a test suite that covers the same paths for the same program, then the attacker will use the test suite output by the black-box testing technique. Hence, we recommend bounding the number of obfuscation transformations applied (to protect a program), by the shortest time needed for a black-box test case generator to produce a test suite that covers all the code of a given binary or to find an input that leads to a certain execution path. For instance, consider a program with a simple control-flow structure such as: `if (x > 127) ... else ...`, where `x` is an unsigned byte input value. Obfuscating this program using multiple layers of obfuscation could certainly make it hard to analyze statically. However, from the point of view of dynamic analysis this program has only 2 paths. Moreover, a black-box testing technique has a 50% probability of finding a test case that covers each of the 2 paths due to the fact that the if-statement on line 1 divides the range of input values into 2 equally sized subranges. Note that this probability changes depending on the range and number of inputs as well as the types of conditional branches inside the code. The effort needed by a black-box test case generator is correlated with the path(s) for which the probability of finding an input is the lowest [31].

2.3 Symbolic and Concolic Execution

Symbolic execution as originally described by King [39] involves simulating the execution of a program by replacing input values of a program with "symbolic" values. As the simulation of the execution progresses constraints are added to "symbolic" values whenever the input is processed. When a branch condition is encountered, the simulation is forked into two paths: one path where the branch condition evaluates to true and the other where it evaluates to false. The premise behind symbolic execution is that all code is available for simulation. However, in practice this may not hold, e.g. for system calls, which execute at OS kernel level.

Concolic execution stands for **conc**rete + symb**olic** execution and it solves the issue of missing code by assigning concrete values to system call arguments and dynamically executing them [32, 12]. The concrete return values and side effects of system calls are then used to continue symbolic execution. Concrete value assignments are obtained by querying a *satisfiability modulo theories* (SMT) solver using the path constraints for a certain path [28]. An SMT solver tries to find an assignment of concrete values to symbolic variables, which will satisfy all path constraints [9]. Other analysis techniques which improve on classical symbolic execution have been developed over the last decade also under the name *dynamic symbolic execution* [11, 45, 50]. In this paper we will use the term symbolic execution to refer to all of the techniques which employ a mix between dynamic analysis and symbolic execution.

2.4 Code Tampering Attacks

Up to this point we have mainly discussed passive analysis attacks, where the attacker may only observe the code of a program statically and/or dynamically. However, for several attacker goals we must also consider active analysis attacks, where the attacker can modify the code statically or while loaded in process memory [43, 42] a.k.a. *tampering* attacks. In particular, the subgoal of identifying self-checking code instructions is necessarily followed by the subgoal of disabling those instructions by modifying the code [43]. Also the goal of bypassing a license check may also be achieved by disabling the conditional instruction(s) which compare(s) the input to the license key [42]. Therefore, an effective obfuscation transformation should not only hamper white-box test case generation, but also hamper active attacks. For instance, data obfuscation techniques such as (cryptographic) hash functions are most effective against passive attackers. However, they are easy to disable by tampering attacks [55]. Hence, we expect control-flow obfuscation techniques [21] to be more successful against active attacks. Basile et al. [10]

Input Size	1 byte	16 bytes
Mean	0.20	4.85
StdDev	0.52	13.54

Depth	1	2
Mean	0.05	2.63
StdDev	0.10	9.55

# Ifs	1	2
Mean	0.01	0.02
StdDev	0.00	0.01

# Input Dep. Ifs	0	1	2
Mean	0.01	0.01	0.02
StdDev	0.00	0.00	0.01

#Loops	1	2
Mean	0.03	3.63
StdDev	0.03	11.42

#Input Len. Dep. Loops	0	1	2
Mean	0.01	0.03	0.05
StdDev	0.01	0.03	0.04

#Input Dep. Loops	0	1	2
Mean	0.02	1.51	1.50
StdDev	0.02	5.90	1.42

Table 1: KLEE execution time (in seconds) of original programs w.r.t. code characteristics of 1st dataset.

and Varia [54] note the strong relation between obfuscation and resistance against tampering attacks. They provide formal models for tamper-resistance.

3. CASE STUDY

In this section we probe existing code obfuscation transformations against state-of-the-art symbolic execution engines. We created 2 datasets of programs[2]. Using the first dataset we analyze the symbolic execution slowdown of obfuscated programs, w.r.t. their unobfuscated counterparts, for the attacker who wishes to attain 100% reachable code coverage (§ 3.2), e.g. to simplify the CFG or to identify and disable self-checks. Our goal in § 3.2 is to determine which obfuscation transformations from 2 freely available obfuscation tools are more resilient against symbolic execution and which transformations should be used in combination with each other. Using the second dataset of programs our goal is to compare different symbolic execution engines (§ 3.3) which simulate the attacker that wants to reach a certain path, e.g. to bypass a license check.

3.1 Obfuscator and Analysis Implementations

Commercial obfuscation tools such as Themida[3], Code Virtualizer[4], VMProtect[5] and ExeCryptor[6] operate exclusively on the Microsoft portable executable (PE) format. However, implementations of state-of-the-art test case generators for PEs such as Microsoft SAGE [34] are not publicly available. Other tools such as KLEE [11], *angr* [50] and Triton [45] have no or limited support for analyzing PEs. BitBlaze Vine [51] and S2E [17] (based on KLEE), support the analysis of PEs, however, as existing work [57] already points out, they have issues analyzing code obfuscated by the commercial tools mentioned above. Therefore, in this work we use the Tigress C Obfuscator [18] and Obfuscator-LLVM [37], which operate at the C source code level, respectively LLVM intermediate representation level. We used KLEE[7] with LLVM version 3.4 and the POSIX runtime provided by its custom *klee-uclibc*[8] and the STP[9] SMT solver [28]. Additionally to using KLEE, in § 3.3 we also use angr version 4.6 and Triton version 0.3.

3.2 Experiments with First Dataset

The first dataset contains 48 manually written C programs consisting of only one function. These programs have between 11 and 24 lines of code including: control-flow statements, integer arithmetic and system calls to `printf`. We have deliberately designed these programs to be small for the following 2 reasons: (1) the size of the obfuscated versions of these small programs increases between 53 and 2918 lines of code and (2) we wanted to run a symbolic execution engine on each program and all of its obfuscated versions without any timeout, 10 times each, in order to check variability of the results. For these 48 programs we varied the following code characteristics (not all possible combinations) in order to increase the heterogeneity of the programs. The values of these code characteristics are listed in the headers of the sub-tables from Table 1: (1) the size of the input of the program measured in bytes; (2) the depth of nested control flow instructions (conditional branches and loops); (3) the total number of if-statements; (4) the number of if-statements dependent on the value of the input (as an integer); (5) the total number of loops; (6) the number of loops depending on the length of the input (in bytes); and (7) the number of loops depending on the value of the input (as an integer).

For the experiments presented in this subsection we used KLEE to generate test suites that cover all the code in each program. We have used a machine with an Ubuntu 14.04 64-bit operating system with 16GB of physical memory and an Intel Core i7-3520M CPU with 4 logical cores each having a frequency of 2.90GHz.

Experiment 1: We ran KLEE on every original program 10 times and we analyzed the average values across these 10 runs. In this and all the of following experiments (including those with angr from § 3.3) the coefficient of variation c_v (i.e. standard deviation divided by the mean) of the symbolic execution time of the same program did not exceed 40% for any program. Moreover, c_v was less than 25% for over 90% of the symbolically executed programs. In other words the execution times for the same program can be considered roughly constant.

Table 1 shows execution time of KLEE for obtaining 100% code coverage on the original (unobfuscated) programs w.r.t. the previous 7 code characteristics. It is important to note that the high standard deviation w.r.t. the mean, is computed over all programs from the dataset and denotes the heterogeneity of the dataset, which is crucial in order to have some degree of generality for our findings. The top-left subtable in Table 1 indicates that the time needed for symbolic execution increases with the size of the symbolic input. The following sub-tables indicate that programs containing only if statements were faster to symbolically execute than programs which contain loops. Moreover, nesting if-statements inside loops and nested loops lead to higher symbolic execution times. Finally, making the branch condition of if-statements or loops dependent on the value of the input also increases symbolic execution time.

Experiment 2: We obfuscated each of the 48 programs

[2]https://github.com/tum-i22/obfuscation-benchmarks
[3]http://oreans.com/themida.php
[4]http://oreans.com/codevirtualizer.php
[5]http://vmpsoft.com/products/vmprotect/
[6]http://www.strongbit.com/execryptor.asp
[7]Rev https://github.com/klee/klee/commit/58f9473
[8]Rev https://github.com/klee/klee-uclibc/commit/a8af87c
[9]Rev https://github.com/stp/stp/commit/3785148

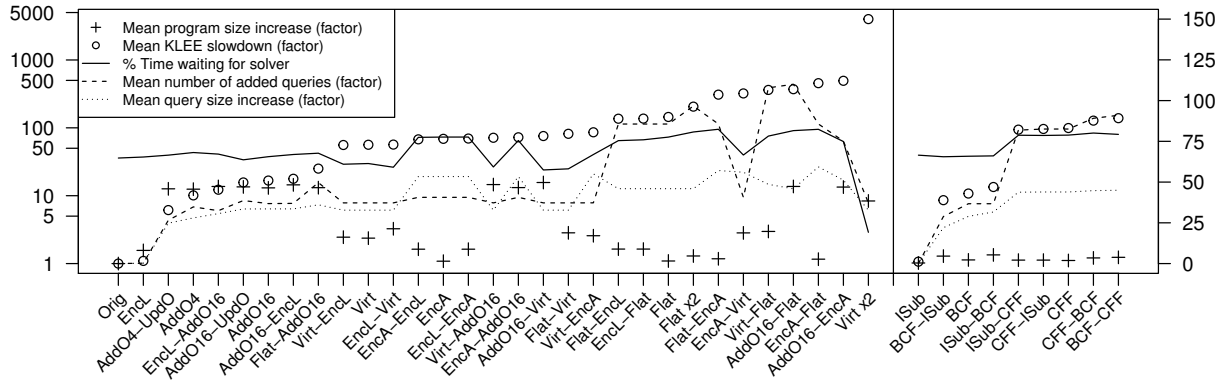

Figure 1: Impact of obfuscation on the KLEE symbolic execution for programs in 1st dataset. X-axis labels to the left of the vertical bar are Tigress transformations; those to the right are Obfuscator LLVM transformations. Right Y-axis is linear and applies only to "% Time waiting for solver" (solid line).

using the following 30 different configurations of Tigress:

1. *EncodeLiterals* (EncL): replaces constant strings and integers by code which dynamically generates these constants during execution.
2. *EncodeArithmetic* (EncA): replaces arithmetic expressions with more complex equivalent expressions.
3. *Flatten* (Flat): transforms the control-flow of a function into a flat-hierarchy of basic blocks that all have the same predecessor and successor basic blocks.
4. *Virtualize* (Virt): transforms a function into an interpreter for a random language L, translates the function's code into L and saves it as bytecode.

5-8. *AddOpaque* (AddO): inserts opaque predicates i.e. branch conditions that are either always true or always false at runtime, but whose value is difficult to analyze statically. *UpdateOpaque* (UpdO): assigns new values to the variables used in opaque predicates, at runtime. We used 4 and 16 AddO, with and without UpdO.

9-28. Ordered combinations of every possible couple of the previous transformations, except for combinations with: *AddO4*, *AddO4-UpdO* and *AddO16-UpdO*.

29-30. *Flatten* and *Virtualize* were each applied 2 times consecutively on the same program.

Note that the only Tigress options used were those indicating the transformation type and the number of opaque predicates. For all other Tigress options we used the default values in order to limit the number of obfuscated programs that we then used as inputs to symbolic execution engines. Also some options of Tigress introduce `goto` instructions and in-line assembly instructions not supported by KLEE.

Each of the 48 programs were also obfuscated using the following 9 different configurations of Obfuscator LLVM:

1. *InstructionSubstitution* (ISub): replaces arithmetic and boolean expressions with a sequence of expressions which evaluate to the same result. This transformation is similar to the *EncodeLiterals* transformation of Tigress.
2. *ControlFlowFlattening* (CFF): similar to the *Flatten* transformation of Tigress.
3. *BogusControlFlow* (BCF): similar to the *AddOpaque* transformation of Tigress.

4-9. Ordered combinations of every possible couple of the previous transformations.

All of the obfuscated programs corresponding to the first dataset were executed using KLEE. We recorded the SMT queries corresponding to every path using the `-write-smt2s` option. The value of the command line option `-sym-arg` indicates the length of the symbolic input argument passed to the program, this was changed accordingly for programs with input arguments having 1 byte and 16 bytes. We executed KLEE on every obfuscated program 10 times and we analyzed the average values across the 10 executions.

We computed the slowdown of symbolic execution of an obfuscated program w.r.t. its unobfuscated counterpart by: dividing the time needed for KLEE to analyze the obfuscated program by the time needed to analyze the corresponding original program. Figure 1 presents the average slowdown (using circles) in ascending order from left to right w.r.t. each employed transformations (X-axis) for the programs obfuscated using Tigress (left of vertical bar in Figure 1) and Obfuscator LLVM (right of vertical bar in Figure 1). Figure 1 also shows: the average increase in program size (plus signs), the percentage of time KLEE spent waiting for the SMT solver to answer all queries (solid line, only line that uses linear Y-axis on right of Figure 1), the average increase in total queries issued to the SMT solver (dashed line) and the average query size measured as number of nodes in the abstract syntax tree of an SMT query (dotted line). We make the following observations from Figure 1.

Observation 1: The EncL and AddO transformations of Tigress (even when employed together or in conjunction with UpdO), have a small impact on the slowdown of symbolic execution compared to the other obfuscation transformations. The reason for this small slowdown is the fact that the additional control-flow instructions introduced by these transformations are not input dependent and they can easily be solved by the SMT solver. This observation also applies to the corresponding transformations of Obfuscator LLVM, i.e. ISub, respectively BCF.

Observation 2: EncL in conjunction with any of the 3 transformations: EncA, Flat and Virt, does not increase the slowdown more than simply employing those transforma-

tions alone. The reason is that EncL only splits constant literals (e.g. from print statements) into a sequence of instructions, which are concretely executed by KLEE. This observation also applies to the corresponding transformations of Obfuscator LLVM, i.e. combining ISub with CFF does not increase the slowdown more than CFF alone.

Observation 3: Resilience increases when applying Virt and Flat after any other transformation. The reason for this is that both of these 2 transformations construct an interpreter-like structure whose complexity is proportional to the size of the given source code. The other 3 transformations of Tigress tend to add more code to the input program, hence applying them before Virt and Flat results in larger interpreter-like structures. This observation also applies to the CFF transformation in Obfuscator LLVM. Moreover, applying Flat after *Virtualize* increases the slowdown.

Observation 4: The average percentage of time spent waiting for the SMT solver to solve path constraints in the original program is equal to 66.35% for the original programs in the first dataset. EncL, AddO, ISub and BCF do not have a significant impact on this value. This is because very few additional queries are added by EncL and ISub. Also KLEE can simplify the majority of path constraints added by AddO and BCF via caching query results, i.e. without calling the SMT solver for each path constraint. On the other hand, EncA, Flat and CFF each cause an increase of about 10% when used separately. EncA and Flat cause an increase of about 15% when used together. However, applying Flat twice does not lead to a 20% increase. Contrarily, Virt decreases the time spent waiting for the SMT solver by about 5% when used alone (despite issuing more and larger queries) and by 5-45% when used in combination with EncL, AddO and Virt.

Observation 5: Flat and CFF on average lead to an increase of 2 orders of magnitude in the number of queries issued by KLEE to the SMT solver. The queries are also approximately 10 times larger than the original queries. This means that the slowdown is due to the fact that KLEE is issuing many more large queries to the SMT solver which is expensive in terms of time. These queries are generated due to branches which are dependent on program input values. Programs which do not contain loops or if statements dependent on input values have no increase in the number of issued queries. An important advantage of flattening is that it has the smallest effect on program size compared to all other employed transformations.

Observation 6: EncA increases the query size 20 times. This suggests that the slowdown for this technique is due to the difficulty of solving the queries and not to the large number of queries as was the case in *Observation 5*. These queries are more difficult to solve due to the non-trivial complexity of the arithmetic expressions added by EncA. Remarkably, the size of the program is affected less by this transformation than by Virt, AddO and even EncL, because the complex expressions added by EncA are not as large as control flow statements added by the other transformations.

Observation 7: Virt tends to reduce the time that KLEE waits for the SMT solver. The reason for this reduction is due to the fact that this transformation adds more instructions to the execution paths, i.e. loading bytecode values and data values for every original statement of the program. Since all instructions have to be interpreted by the symbolic execution engine, this translates into higher slowdown. Applying Virt multiple times yields multiple levels of interpreter *fetch-decode-dispatch* instructions, hence the large slowdown when virtualization is applied twice. Similarly to flattening, Virt induces a higher slowdown on programs with loops or if-statements depending on program inputs.

Experiment 3: KLEE outputs an SMT file for every path that it finds in the program it analyzes. These SMT instances contain the path constraints necessary to generate a test case that leads to the corresponding path. We compared the SMT instances output by KLEE for all programs in the dataset (including the original programs).

Observation 8: The sets of SMT instances corresponding to each obfuscated version of a program are practically identical to the set of SMT instances of the original (unobfuscated) program, except for EncA, which were significantly larger. For all transformations (except EncA) only small differences occur such as using shifting and adding instead of multiplication. This is due to the fact that KLEE uses concrete values for instructions added by the majority of obfuscation transformations employed in this case-study. The concrete values can be simplified away from path constraints even without calling the SMT solver.

Observation 9: The time the symbolic engine waits for the SMT solver to find an answer to the query accounts for almost all the increase in effort added by obfuscating a program using EncA. Therefore, one possibility for improving obfuscation is to make these expressions more complex and difficult to solve by applying the EncA transformation multiple times to the same program. However, such an obfuscation could be bypassed by an attacker who knows all the substitution rules that the EncA transformation can use and then simply applies them backwards. Rolles [44] has developed a method that could automatically extract such substitution rules from programs by employing SMT solvers. In general, the complexity of SMT instances can be increased by non-linear transformations such as those employed in cryptographic hash functions (see Section 5.2).

Observation 10: None of the obfuscation transformations insert additional paths dependent on input values to the program, i.e. the sizes of the sets of SMT instances corresponding to each obfuscated version of a program have the same size as the set of SMT instances of the original (unobfuscated) program. This may be counter intuitive since AddO and BCF insert additional if statements. Since these if statements are not input dependent and always have the same truth value, the SMT solver is able to eliminate the dead branches of these if-statements, i.e. the symbolic execution engine will not analyze the code in those dead branches.

3.3 Experiments with Second Dataset

The second dataset contains over 5000 C programs consisting of a main function and another function f_i randomly generated by the *RandomFuns* feature of Tigress. Each program from the second data set computes a different function $f_i(argv[1])$ and outputs some integer number. The output of the function is then compared to a hard coded integer value equal to $f_i(12345)$, as shown in Listing 1. If the comparison succeeds then the program prints a distinctive message on the standard output. This comparison resembles a license check. Finding an input value that passes this comparison is harder for a white-box test case generator to find, than an input that would fail the comparison.

For the experiments described in this subsection we used

Listing 1: Example program from 2nd dataset.

```
1 // f_i("12345") = 0x76543210
2 if (f_i(argv[1]) == 0x76543210)
3   printf("win");
```

Data Types	char	short	int	long
Mean	1.32	9.95	13.41	13.91
StdDev	0.98	6.48	7.86	8.34

Loop Bound	Constant	Bounded Input	Input
Mean	8.45	8.43	10.62
StdDev	8.07	7.28	9.65

Operators	Bitwise	Simple Arith.	Harder Arith.	All
Mean	4.74	8.91	9.97	11.23
StdDev	5.60	6.81	8.60	8.60

Table 2: KLEE execution time (seconds) on original programs w.r.t. code characteristics of 2nd dataset.

a machine with more cores to enable running multiple symbolic executions in parallel. The machine uses the Ubuntu 14.04 64-bit operating system and it has an Intel Xeon E5-1650v2 CPU with 12 logical cores each running at 3.50GHz and 64GB of physical memory. For second dataset of programs we varied the same code characteristics as for the manually written programs in the previous dataset. Additionally we also varied the following characteristics to see if they influence the analysis of symbolic execution engines:

1. The data types of variables: char, short, int and long.
2. The types of bounds placed on loops: integer constants, integer input values bounded via modulo a small constant and unbounded integer input values.
3. The types of operators used inside statements: simple arithmetic (i.e. addition and subtraction), harder arithmetic (which also includes multiplication, modulo and division), bitwise (i.e. logical and, or, xor and bit shifting) and a combination of all operators.

Experiment 4: The impact of these 3 code characteristics on the execution time of KLEE given the original (unobfuscated) programs can be seen in Table 2. As was the case with the input size in Table 1, the symbolic execution time increases with the size (ranges) of the data types. Different types of bound conditions placed on loop statements cause a mild difference on symbolic execution time, i.e. if the loop iterates a constant number of times, then the symbolic execution engine will execute faster on average than if the number of loop iterations depends on the program input. Finally, the type of operators used by the program have an important impact on symbolic execution, because these operators are used by path constraints which are issued as queries to the SMT solver. We notice that bitwise operators are easier to solve than arithmetic operators. The type of arithmetic operators does not cause a large difference in symbolic execution. However, harder arithmetic tends to be slower to solve than simple arithmetic. More importantly, combining all operators seems to have an additive effect w.r.t. the time taken to solve path constraints.

Experiment 5: We have obfuscated the f_i functions with 5 obfuscation transformations from the Tigress tool: AddO, EncA, EncL, Flat and Virt. We only chose these 5 transformations, due to the fact that Obfuscator LLVM transformations are very similar to AddO, EncL and Flat. For this

	KLEE			angr		
	Median	Mean	StdDev	Median	Mean	StdDev
AddO16	0.97	1.03	0.26	1.72	2.25	2.49
EncA	1.14	1.21	0.37	1.39	1.79	1.90
EncL	0.98	0.99	0.22	1.40	2.22	4.60
Flat	1.15	1.22	0.44	3.77	4.45	2.85
Virt	1.53	2.08	1.27	7.32	8.85	5.01

Table 3: Symbolic execution slowdown on programs obfuscated using Tigress, relative to unobfuscated counterparts from 2nd dataset.

experiment we used KLEE and *angr* as symbolic execution engines and we let them run until they found the path in the program that prints a distinctive message on the standard output or the timeout of 1 hour is reached. When this path is entered we know that the check guarding that path has been bypassed by the symbolic execution engine. We did not use angr in previous experiments because (as far as the authors are aware) angr does not aim to achieve 100% code coverage, as opposed to KLEE. Note that we have also tried to employ the Triton symbolic execution engine [45] on the obfuscated programs for both datasets. However, Triton crashed when symbolically executing programs obfuscated using Flat and Virt due to insufficient memory. Triton transforms each assembly instruction into a sequence of SMT constraints, which increases directly proportional to the execution trace, which is large for programs obfuscated with Flat and Virt.

Table 3 shows the median, mean and standard deviation of symbolic execution slowdown on programs obfuscated from the second dataset w.r.t. their unobfuscated counterparts. The slowdown is computed as the time needed to symbolically execute an obfuscated program until the path in the program that prints the distinctive message on the standard output is found, divided by the time need to symbolically execute the unobfuscated version of the program to find the corresponding path. The median and standard deviation were taken across 12713 obfuscated programs successfully analyzed by the KLEE and angr within the 1 hour time limit. We make the following observations using Table 3.

Observation 11: KLEE incurs a lower slowdown than angr for all of the 5 obfuscation transformations employed in this experiment. Therefore, we conclude that KLEE is the worst case attacker for the obfuscation transformations we have employed in this paper. Note that KLEE also has limitations, e.g. it does not support `goto` instructions or in-line assembly in C programs. However we see this as a technical, not a fundamental limitation.

Observation 12: The slowdown of finding the path that prints a distinctive message ("win") is much lower than the slowdown for covering all reachable code (which was the goal of the attacker in § 3.2). This is expected since the symbolic execution engine may discover that particular path before covering all reachable code.

3.4 Summary and Threats to Validity

In § 3.2, we have symbolically executed a set of programs obfuscated with 39 different configurations of 8 transformations from 2 obfuscation tools. We generated 100% code coverage test suites that would be used by an attacker who aims to simplify the CFG of an obfuscated program. The results indicate that all obfuscation transformations can be broken with different computational effort, which also depends on

Listing 2: *Range divider* with 2 branches

```
if (x > 42)
  z = x + y + w
else
  z = (((x ^ y) + ((x & y) << 1)) | w) +
      (((x ^ y) + ((x & y) << 1)) & w);
```

Listing 3: Program with loop

```
unsigned char *str = argv[1];
unsigned int hash = 0;
for(int i = 0; i < strlen(str); str++, i++) {
  hash = (hash << 7) ^ (*str);
}
if (hash == 809267) printf("win\n");
```

code characteristics of the original program. EncL and ISub are not effective against symbolic execution. AddO and BCF have a mild effect on the slowdown. EncA slows down symbolic execution due to the larger size of SMT queries. Flat and CFF slow down symbolic execution due to the larger number of SMT queries issued to the solver. Virt slows down symbolic execution due to the high number of fetch-decode-dispatch instructions added. The results also indicate in which order obfuscation transformations should be applied to increase effectiveness. None of these obfuscation transformations add input-dependent paths to the programs.

In § 3.3 we have symbolically executed another set of programs obfuscated with 5 representative transformations from Tigress used in § 3.2. Transformations from Obfuscator LLVM were not used in § 3.3 because they had similar effects as their corresponding transformations from Tigress. In this experiment we compared the performance of different symbolic execution engines to find test cases that lead to a certain (difficult to reach) path in the obfuscated programs. We observed that KLEE is most effective followed by angr.

We do not know if these results generalize for all possible programs. However, we believe that they have some degree of generality due to the heterogeneity of our datasets and the intuitive explanations we provide in our observations.

4. PROPOSED OBFUSCATION

The proposed obfuscation transformations are inspired by observation 10 in Experiment 3 from § 3.2, i.e. branch instructions added by control-flow obfuscation transformations do not depend on program inputs. Therefore, we propose making branch instructions added by control-flow obfuscation transformations dependent on program inputs to increase the slowdown of symbolic execution. Making such branch instructions input dependent may or may not be effective for existing obfuscation transformations. For instance, making the opaque predicates p added by Tigress input dependent, causes KLEE to issue a query on the branch condition corresponding to p. However, the SMT solver always determines that p or $\neg p$ is unsatisfiable and does not analyze the code in unfeasible paths. In the following we propose two obfuscation transformations which introduce feasible paths in the original program.

4.1 Range Dividers

Our first proposal is an obfuscation transformation called *range divider*. *Range dividers* are branch conditions that can be inserted at an arbitrary position inside a basic block, such that they divide the input range into multiple sets. In contrast to opaque predicates, *range divider* predicates may have multiple branches, any of which could be true

Listing 4: Program from Listing 3 obfuscated with *range divider*

```
unsigned char *str = argv[1];
unsigned int hash = 0;
for(int i = 0; i < strlen(str); str++, i++) {
  char chr = *str;
  if (chr > 42) {
    hash = (hash << 7) ^ chr;
  } else {
    hash = (hash * 128) ^ chr;
  }
}
if (hash == 809267) printf("win\n");
```

Listing 5: Program from Listing 3 obfuscated with maximum number of branches of *range divider*

```
unsigned char *str = argv[1];
unsigned int hash = 0;
for(int i = 0; i < strlen(str); str++, i++) {
  char chr = *str;
  switch (chr) {
    case 1: hash = (hash << 7) ^ chr;
      break;
    case 2: // obfuscated version of case 1
      break;
    ...
    default: // obfuscated version of case 1
      break;
  }
}
if (hash == 809267) printf("win\n");
```

and false depending on program input. This will cause a symbolic execution engine to explore all branches of a range divider. In order to preserve the functionality property of an obfuscator, we use equivalent instruction sequences in all branches of a *range divider* predicate, as illustrated in Listing 2. To prevent compiler optimizations from removing *range divider* predicates, due to equivalent code in their branches, we employ software diversity (on the code of every branch) via different obfuscation configurations, e.g. in Listing 2 we used *EncodeArithmetic* on the *else* branch. We have experimented with all optimization levels of LLVM *clang* and none of them remove *range divider* predicates if their branches are obfuscated. On the downside, *range dividers* increase the size of the program proportionally to the total number of branches.

The effectiveness of a *range divider* predicate against symbolic execution depends on: (1) number of branches of the predicate, denoted ρ and (2) the number of times the predicate is executed, denoted τ. More specifically, its number of paths increases according to the function: ρ^{τ}. For example, consider the program from Listing 3, which computes a value (`hash`) based on its first argument (`argv[1]`) and outputs "win" on the standard output if this value is equal to 809267. It has an execution tree with $2 \times$ `strlen(argv[1])` paths. The program in Listing 4 is obtained by obfuscating the program from Listing 3 using *divide range* predicate with $\rho = 2$ branches. The resulting program has $2^{\texttt{strlen(argv[1])}}$ paths, because the predicate is executed $\tau =$ `strlen(argv[1])` times. We can further increase the number of paths by adding more branches to the *divide range* predicate from Listing 4. However, the number of possible branches is upper-bounded by the cardinality of the type of the variable used in the range divider. In the example from Listing 4 the maximum number of branches is 256 because variable `chr` is of type `char`. Therefore, the maximum number of branches in this example is achieved by a *switch*-statement

Listing 6: Point function program

```
1   if (argv[1][0] == '1' &&
2       argv[1][1] == '2' &&
3       argv[1][2] == '3' &&
4       argv[1][3] == '4' &&
5       argv[1][4] == '5') printf("win\n");
```

with 256 cases as shown in Listing 5. This effectively results in a branching factor of $\rho = 256$ for each iteration of the loop. Therefore the number of paths is $256^{\texttt{strlen(argv[1])}}$.

The symbolic execution slowdown induced by *range dividers* is caused by: (1) the increase in the number of paths, but also (2) the type of obfuscation transformations applied on each branch of each *range divider*. Due to lack of space we do not provide an evaluation of *range dividers* here. Instead, we turn to a related obfuscation transformation in § 4.2, which is even stronger against symbolic execution.

4.2 Input Invariants

The *divide range* obfuscation transformation proposed in § 4.1 may induce a high increase in the effort needed by symbolic execution engines. However, this depends on the code which is being obfuscated. If the code does not include any loops (e.g. the program from Listing 6), then obfuscating with *range dividers* will not induce a significant slowdown of symbolic execution attacks. Therefore, in this section we propose obfuscation transformations which are able to obfuscate even the simplest code, however, with the cost of changing the exact input-output semantics of the program. That is, we deliberately violate the *functionality* property of the obfuscator definition of Barak et al. [8]. The definition in [8] states that if $O(P)$ is the obfuscated version of program $P : D \rightarrow R$, where $D, R \subset \{0,1\}^*$ are the input domain, respectively output range of the program, then $\forall i \in D : P(i) = O(P)(i)$. However, Barak et al. [8] also define an *approximate obfuscator* as a transformation for which the *functionality* property holds with high probability. Similarly to an approximate obfuscator our obfuscation approach relaxes the *functionality* property, but does so in a different way. In our approach the *functionality* property only holds for the set of inputs that satisfy the *input invariants* (i.e. predicates over inputs), specified by the user to the obfuscation engine. For all other input values the behavior of the program is undefined. With this obfuscation approach, we are essentially extending the input domain and output range of a program, i.e. $O(P) : D' \rightarrow R'$, where $D \subseteq D'$ and $R \subseteq R'$. In fact one could imagine the extensions go even further, allowing $O(P)$ to *fail* in different ways than P, such as P crashing on bad inputs, while $O(P)$ entering an infinite loop on the same bad inputs, or producing the wrong results, etc. We believe this idea has very interesting implications for future implementations of obfuscation transformations.

We have implemented this transformation on top of the *Virtualize* transformation of Tigress[10]. We picked *Virtualize* because it was the strongest transformation in our case study, however, note that this idea could be applied to other transformations as well. We encoded the bytecode generated by *Virtualize* using the input invariant (specified as an argument of the obfuscation transformation), as the key. If the input value given by the user satisfies the invariant, then the bytecode is correctly decoded and the program executes like its unobfuscated counterpart, otherwise its behavior is undefined. By using the invariant as a decoding key we multiply the size of the search space for symbolic execution by the cardinality of the range of possible key values (for obfuscations different from virtualization, polynomial increases seem possible). A user can specify the input invariants using: (1) the position of the argument in the list of arguments, (2) the type of the argument (integer or string) or its length and (3) the exact value or the interval of possible argument values. Note that different invariant types lead to keys with different cardinalities. The invariants with the highest cardinality keys are those that specify an integer or string argument with an exact value.

The effectiveness of this transformation against symbolic execution engines is higher than any other transformation we have employed in our case-study. To illustrate its effectiveness, we have chosen a program consisting of a single if-statement shown in Listing 6, because it is representative of the simplest possible code structure that one may want to protect against symbolic execution. We obfuscated this program using our modified *Virtualize* transformation with the invariant that the input is equal to 12345 and executed both the original program from Listing 6 and its obfuscated counterpart using KLEE. The point function from the program in Listing 6 was analyzed in approximately 500 milliseconds. While attempting to run the symbolic execution engine uninterrupted, similar to experiment 2 from § 3.2, we resorted to stopping the analysis of the obfuscated program after it ran for 1 week. However, we note that the test suite that would find the path that prints "win" (the goal of experiment 5 in § 3.3) was found in approximately 4980 seconds, which is still a slowdown by 4 orders of magnitude w.r.t. the unobfuscated counterpart. Contrast this with the smaller slowdown factors from Table 3.

5. RELATED WORK

Our work involves contributions related both to the characterization of obfuscation strength and to new obfuscation approaches, as spelt out in the following.

5.1 Characterizing Obfuscation Strength

Ceccato et al. [15] characterize the strength of obfuscation transformations by potency against human-assisted attacks, e.g. step-by-step debugging. This involves user studies where test subjects are asked to perform some tasks (e.g. bypass a check or recover information) on code obfuscated with a limited number of transformations. Such user-studies are inherently biased by small sets of test subjects. Moreover, test subjects are generally bachelor or master students of computer science, which are seldom experienced in reverse engineering obfuscated code. Nevertheless, such works are important for measuring potency against human-assisted attacks; our work is complementary to this approach.

Anckaert et al. [3] characterize the resilience of code obfuscation transformations via code complexity metrics, e.g. McCabe cyclomatic complexity, knot count etc. This approach is typically static and does not work reliably for code obfuscated with transformations which modify the code at runtime (e.g. virtualization, self-modifying code). Our work is different to this approach because in § 3 we indicate that code characteristics that are most important for slowing down symbolic execution, (such as the number of control-flow statements dependent on input and their depth) are not increased by applying a set of obfuscation transforma-

[10]http://tigress.cs.arizona.edu/

tions, even though code complexity metrics are increased.

Dalla Preda [23] models attacks against obfuscation transformations as abstract domains expressing certain properties of program behaviors. Since obfuscation transformations are characterized by the most concrete preserved property, the complete lattice of abstract domains allows comparing obfuscation transformations with respect to their potency against various attackers. Therefore, an obfuscation transformation is either effective against an attacker or not, regardless of the difference in effort needed to deobfuscate programs obfuscated with different transformations. Our work provides a more fine grained characterization of the resilience of obfuscation transformations w.r.t. to the effort required by the attacker to deobfuscate a program.

5.2 Anti-Symbolic Execution Obfuscations

Anand et al. [1] indicate 3 fundamental issues of symbolic execution: (1) path explosion, (2) path divergence and (3) complex constraints. Therefore, works that propose obfuscation techniques against symbolic execution focus on exploiting at least one of these fundamental issues.

Path Explosion: Wang et al. [56] propose an approach based on an unsolved mathematical problem, which involves only linear operations on integers, called the Collatz conjecture [30]. This obfuscation implies adding a loop bounded by a symbolic value to an existing program. Such a loop would generate a path explosion for the symbolic execution engine, however, executing it dynamically, it will always converge to a fixed known value, i.e. 1 in the case of the Collatz conjecture. Our work also proposes an approach that causes path explosion, however, in contrast to Wang et al. [56], our approach is not based on unsolved mathematical conjectures.

Path divergence: Yadegari and Debary [57], Sharif et al. [49] and Cavallaro et al. [13] show that converting explicit control-flow into implicit control-flow hampers white-box test case generators based on taint-analysis and symbolic execution. Path divergence refers to situations where the symbolic execution engine cannot compute precise path constraints from the program code. This leads to a divergence between generated tests and the actual program paths.

The key to transforming explicit to implicit control flow is using symbolic variables for computing an address where the program jumps to unconditionally. This transformation removes any comparisons between the symbolic variables and constants used by symbolic execution engines. In the case of such obfuscation transformations one may employ heuristics based on knowledge about the architecture of the CPU [57].

Another type of obfuscation transformation which is known to cause path divergence is dynamically modifying code [4]. Such code cannot be analyzed statically, because its static image changes at runtime while the code is executing. If the value of the instructions that are being dynamically generated depend on input values, then the symbolic execution engine must guess the right value of the instruction that is to be executed, which is difficult. The drawback of dynamically modifying code is that it affects the input output behavior of the program and opens the door to remote attacks. Since most software developers are more concerned about remote attacks than automatic analysis attacks, they would avoid using such transformations and resort to obfuscation transformations which do not induce this risk. Therefore, in this work we focus on obfuscation transformations which do not introduce remote attacks.

Complex Constraints: Applying cryptographic hash functions to equality checks based on input values is a type of data obfuscation transformation, which is problematic for symbolic execution. In particular, the SMT solvers used by symbolic execution engines are known to have practical limitations w.r.t. inverting cryptographic hash functions [41, 49]. However, we note that cryptographic hash functions are based on large look-up tables containing random numbers which are publicly known and easy to locate in code. Therefore, we argue that an active attacker as described in § 2.4 will be able to locate and disable explicit checks which use hash functions on input values. Implicit checks cannot be disabled, however they may cause the application to crash which is undesirable for many developers or even worse they may open the door for remote attacks as discussed in the previous paragraph. Moreover, note that hash functions are only applicable for equality comparisons and therefore range comparisons need to be protected in a different way. In this paper we discuss obfuscation transformations that hide the location of checks via control-flow obfuscation.

6. CONCLUSIONS

We have discussed why test case generation is a common subgoal of three frequent attacker goals: simplifying the CFG; identifying and disabling integrity checks; and bypassing license checks. Our empirical case-study indicates that for the datasets of programs used in this paper, a subset of existing obfuscation transformations are weak against white-box test case generation via symbolic execution. Since these datasets are heterogeneous, as indicated by the high standard deviations in Tables 1 & 2, we believe these results to generalize to other programs.

Symbolic execution has several limitations when applied to large software programs (§ 5). However, we note that symbolic execution is currently a highly active field of research and has been successfully applied for finding bugs in Microsoft Windows 7 [33] and it is being used by several teams in the DARPA Cyber Grand Challenge for automated exploit generation [5]. Moreover, several deobfuscation techniques rely on symbolic execution [47]. Furthermore, obfuscation is often applied only to parts of software to minimize performance impact [19], hence attackers may isolate and symbolically analyze smaller parts of the software [2].

We also proposed an obfuscation approach that improves resilience against symbolic execution. We have implemented this approach and empirically verified that it increases slowdown of symbolic execution by 4 orders of magnitude w.r.t. the original (unobfuscated) program. We offer our implementation freely to members of academia and industry.

In future work we wish to extend this study using more datasets of programs, more obfuscation implementations and more symbolic execution engines. Additionally, we wish to measure the resilience of obfuscation against active attacks (see § 2.4). We believe that this empirical study may be useful in developing a model that can accurately predict the resilience of different obfuscation techniques w.r.t. automated analysis attacks such as symbolic execution. For this purpose we intend to use machine learning algorithms for training a model and then test its prediction accuracy. Finally, we also plan to perform analytical studies and offer a formal definition of practical obfuscation.

Acknowledgements: We thank Saumya Debray and Martín Ochoa for their valuable insights and feedback. Coll-

berg was supported by National Science Foundation grants 1525820 and 1318955.

7. REFERENCES

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381. Springer, 2008.

[3] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In *Proceedings of the 2007 ACM workshop on Quality of protection*, pages 15–20. ACM, 2007.

[4] D. Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, pages 317–333. Springer, 1996.

[5] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[6] S. Banescu, M. Ochoa, and A. Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 45–51. IEEE, 2015.

[7] S. Banescu, A. Pretschner, D. Battré, S. Cazzulani, R. Shield, and G. Thompson. Software-based protection against changeware. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 231–242. ACM, 2015.

[8] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology CRYPTO 2001*, pages 1–18. Springer, 2001.

[9] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[10] C. Basile, S. Di Carlo, T. Herlea, V. Business, J. Nagra, and B. Wyseur. Towards a formal model for software tamper resistance. In *Second International Workshop on Remote Entrusting (ReTtust 2009)*, volume 16.

[11] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM. 00041.

[13] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008.

[14] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. The effectiveness of source code obfuscation: an experimental assessment. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 178–187. IEEE, 2009.

[15] 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.

[16] H. Chang and M. J. Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2001.

[17] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[18] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328, New York, NY, USA, 2012. ACM.

[19] C. Collberg and J. Nagra. Surreptitious software. *Upper Saddle River, NJ: Addision-Wesley Professional*, 2010.

[20] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[21] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[22] 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.

[23] M. Dalla Preda. *Code obfuscation and malware detection by abstract interpretation*. PhD thesis, University of Verona, 2007.

[24] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Third IEEE International Conference on Software Engineering and Formal Methods.*, pages 301–310. IEEE, 2005.

[25] C. Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[26] 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.

[27] M. Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, pages 7–16. ACM, 2010.

[28] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.

[29] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proc. of the 54th Annual Symp. on Foundations of Computer Science*, pages 40–49, 2013.

[30] L. E. Garner. On the Collatz 3n + 1 algorithm. *Proceedings of the American Mathematical Society*, 82(1):19–22, 1981.

[31] I. P. Gent, E. MacIntyre, P. Prosser, T. Walsh, et al. The constrainedness of search. In *AAAI/IAAI, Vol. 1*, pages 246–252, 1996.

[32] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005.

[33] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[34] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[35] Y. Guillot and A. Gazet. Automatic binary deobfuscation. *Journal in computer virology*, 6(3):261–276, 2010.

[36] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Security and privacy in digital rights management*, pages 141–159. Springer, 2002.

[37] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In B. Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[38] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *19th Working Conference on Reverse Engineering (WCRE)*, pages 61–70, Oct 2012.

[39] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[40] McAfee. McAfee Labs Threats Report. Technical Report March, 2016. http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf.

[41] I. Mironov and L. Zhang. Applications of sat solvers to cryptanalysis of hash functions. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 102–115. Springer, 2006.

[42] G. Naumovich and N. Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, (7):64–71, 2003.

[43] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. Identifying and understanding self-checksumming defenses in software. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 207–218. ACM, 2015.

[44] R. Rolles. Program Synthesis in Reverse Engineering. http://www.nosuchcon.org/talks/2014/D1_01_Rolf_Rolles_Program_Synthesis_in_reverse_Engineering.pdf, 2014. NoSuchCon 2014, Accessed:2016-05-24.

[45] F. Saudel and J. Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

[46] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)*, 49(1):4, 2016.

[47] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[48] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 94–109, May 2009.

[49] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[50] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[51] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information systems security*, pages 1–25. Springer, 2008.

[52] Symantec Corporation. Internet Security Threat Report 2016. Technical Report April, 2016. https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf.

[53] S. Udupa, S. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering*, 2005.

[54] M. Varia. *Studies in program obfuscation*. PhD thesis, School of Computer Science, Tel Aviv University, 2010.

[55] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[56] Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In *Computer Security–ESORICS 2011*, pages 210–226. Springer, 2011.

[57] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, ser. CCS*, volume 15, pages 732–744, 2015.

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