Analysis of Obfuscated Code with Program Slicing

Mahin Talukder
Dept. of Computing and Engineering
University of East London
London, UK
u1539655@uel.ac.uk

Syed Islam
Dept. of Computer Science
University College London
London, UK
s.islam@cs.ucl.ac.uk

Paolo Falcarin
Dept. of Computing and Engineering
University of East London
London, UK
falcarin@uel.ac.uk

***Abstract*—In Man-At-The-End (MATE) attacks, software apps run on a device under full control of the attackers: they can violate the intellectual property of the app by means of malicious reverse engineering, software piracy, and software tampering. Obfuscation is a technique that is widely adopted by developers to mitigate this problem. Obfuscation increases complexity of software code, by obscuring the structure of code and data in order to thwart the reverse engineering process. However, it is possible to reverse engineer obfuscated code with time, determination and the right tools. In general, there is no accepted methodology to determine the strength of obfuscated code; however *resilience* is often considered a good metric as it indicates the percentage of obfuscated code that cannot be removed by automated de-obfuscation tools. We introduce a novel approach to measure the *resilience* of obfuscated C code using program slicing. Given a variable of interest, that might be part of a code region used to manipulate a crypto key or a license number, program slicing can mimic the attacker behaviour by trying to remove the code unrelated to that variable, acting as a new type of de-obfuscator.**

***Index Terms*—Program slicing, Code obfuscation, ORBS, reverse engineering, resilience, de-obfuscation, software security, MATE attacks**

I. Introduction

The latest BSA Global Software Survey on Software Piracy [1] states 37% of software installed on computers around the world in 2017 are not appropriately licensed, amounting to $46 billion in losses. When software apps run on a device under full control of the attackers they can be victims of software piracy, usually achieved through Man-At-The-End (MATE) attacks [2], where attackers use reverse engineering tools to analyse and modify the code to use it in unauthorised ways: e.g. removing license checks or illegally duplicating copyrighted material. Code Obfuscation is a technique that is widely adopted by developers to mitigate this problem by increasing the complexity of the code, in order to thwart the reverse engineering process .

For more than a decade, security and compiler optimisation researchers have proposed various techniques of code obfuscation [3] to prevent code from being analysed, and tamper-proofing [4] to detect the execution of tampered code; other techniques like software watermarking [5], fingerprinting [6] have been developed to bind authorship to the code so that violations could be used as evidence in legal courts. The software industry and large companies like Google [7], Microsoft and Intel [8] have also invested significantly in these technologies. However, secrets kept hidden within code will eventually be discovered by a sufficiently resourced and determined hacker. Researchers aim to stop the majority of hackers and to slow down the more expert ones to a point in which an attack would be no longer useful.

Code Obfuscation preserves the original functionality of a program while adding redundant information to the code in order to make it unintelligible for humans, and to slow down or crash automated program comprehension tools so that the result of program analysis is wrong or incomplete [9]. There is currently no agreement on the methodology to determine which obfuscation technique provides the best possible software protection. One of the more recent metrics used to assess the effectiveness of obfuscation is the concept of *resilience*, defined as the percentage of obfuscated code that cannot be removed by automated de-obfuscation tools, and it clearly depends by the type of tool and the target language, e.g.: interpreted or compiled code, bytecode or native code. In this paper, we present a novel approach to determine the *resilience* of native code (generated from C language) using program slicing. Mark Weiser introduced program slicing with a view to understanding behaviour of a program's constructs [10]. The technique was built based on how programmers debug their programs by looking at subsets of the program that are of interest; for example, where a variable is printing incorrect results. This is very similar to what an attacker is trying to do when attempting to reverse engineer a piece of software, and then we propose to apply program slicing to estimate the *resilience* of software obfuscation. In the rest of the paper, Section II provides background to the problem in terms of code obfuscation and introduces program slicing. Section III, presents the methodology and toolchain. Section IV describes the experimental setup and Section V presents results from the study. Section VI discusses related works and section VII presents experimental data. Finally, Section VIII provides a summary of the current work and ideas for future research.

II. Background

A. Code Obfuscation

Code obfuscation is the most viable method for preventing reverse-engineering [11]: the goal is to make the code difficult to understand by using a software tool called *obfuscator*, a special compiler that can perform binary-to-binary, source-to-binary or source-to-source code transformations. As we aim to determine *resilience* of the code generated by the obfuscator, we have chosen source-to-source code obfuscation to have more control on the compiler options, and monitor code changes through our experiments, while in source-to-binary and binary-to-binary obfuscators many decisions made by the tool are neither visible nor configurable.

Lexical transformations, Control flow transformations, Data transformations, Anti-disassembly, Anti-debugging, and Encryption are the types of obfuscation that are commonly used by obfuscators [5], offered by companies like Semantic Designs [12], Irdeto [13], and Stunnix [14], while Tigress [15] is a well-known open source obfuscator with different kinds of protection techniques and is widely used for research. Tigress supports three advanced types of transformations:

- *Virtualization* – transforms a function into an interpreter
- *Jitting* – transforms a function into one that generates its machine code at runtime.
- *JitDynamic* – transforms a function into one that continuously modifies its machine code at runtime.

In addition to these, Tigress implements other common obfuscation transformations, such as: *Control flow flattening*, *Function splitting*, *Function merging*, *Argument randomization*, *Control flow splitting with opaque predicates*, *Encoding of literals*, and, *Data and arithmetic transformation*. For the experiments in this paper, we employ *Virtualization* along with *Control-flow flattening* and *Function splitting* as they are complex and commonly used obfuscation transformations. Each of these transformations has a certain number of parameters which are used to create many obfuscated code variants.

B. Program Slicing

Program slicing is the computation of a set of programs statements, that may affect the value of a variable at some point of interest [16]. It is a decomposition technique that extracts statements relevant to a particular computation from a program [17]. A slice could be regarded as the mental abstraction people make when they are debugging a program. A slice consists of all the statements of a program that may affect the values of some variables at some point of interest referred to as a slicing criterion [10].

A slice is computed using either static information (without program execution) known as a static slice, or dynamic information (execution of a program) known as a dynamic slice. Static slicing seeks to find an executable subset of a program's statement that exhibits the same behaviour of a specified variable at a specific location as the original program for all possible inputs. On the other hand, dynamic slice preserves the behaviour of slicing criterion only with respect to specific input [18].

Observation based slicing was recently introduced by researchers as an alternative to dependence-based program slicing. Observation based slicing (ORBS) [18] uses a *delete–execute–observe* approach where a statement is deleted, then the program is executed and an observation is made on whether the projected trajectory of the slicing criterion is changed. If no changes are observed, the deleted statement is excluded from the slices. Conversely, if a change is observed, the deleted statement is included in the slice. This process is iterated until no further statements can be excluded from the slice. ORBS was introduced with a view to slice heterogeneous programs consisting of components written in different programming language and perform slicing that includes binary components or external libraries. ORBS is designed to reduce the length of a program to an abstract subset requiring less human effort to understand the program. Our approach uses ORBS to determine the resilience of obfuscated code.

III. Methodology and Toolchain

In this section, we described the methodology and our Toolchain for the experimentation. Our current Toolchain integrates Tigress as obfuscator and ORBS frameworks into an automated Toolchain that can create multiple obfuscated *variants* of a *target* program, and generate the program slices for both the original target program and its obfuscated variants. The approach can be adopted with any source-code to source-code obfuscator and different programming languages (instead of C language used by Tigress) as ORBS frameworks is programming language independent. Comparison of the slices from these data points help us to determine and understand the effectiveness of obfuscation techniques used to produce each variant.

Figure 1 presents a graphical representation of our Toolchain, which executes as a process made of different phases, as depicted in the diagram. In the beginning (Stage 1) the original target program (Program *P1*) is instrumented to mark the slicing criteria - that is, the points of interest, so that ORBS slicer is able to subsequently extract dependencies for these points. We selected a point of interest from each program to indicate a key functionality that software vendors may want to protect from Man-At-The-End (MATE) attack. This scenario is similar to the approach where an attacker tries to reverse engineer the code to identify a piece of code containing sensitive information. It focuses on a piece of code that may be of interest and identifies all the associated code constructs. The slice of the original target program from this stage is later used as ground truth to understand the effectiveness of the obfuscation.

The next stage involves using Tigress to produce several obfuscated variants of the original program. We have generated multiple variants of the same target program by running Tigress with different combinations of parameters. These variants are marked as *P1_V1* to *P1_Vn*, where n is the number of variants created. The number of versions generated for every target program may not be the same as some combinations of parameters do not form a valid configuration for Tigress or they might not work for a particular program structure. Obfuscated code size is typically bigger than the original code, because of the additional bogus code included to hide the actual functionality and to make it more difficult to read and understand the program.

In the third stage, the toolchain mimics what an attacker would do once identified a point of interest in the code: find

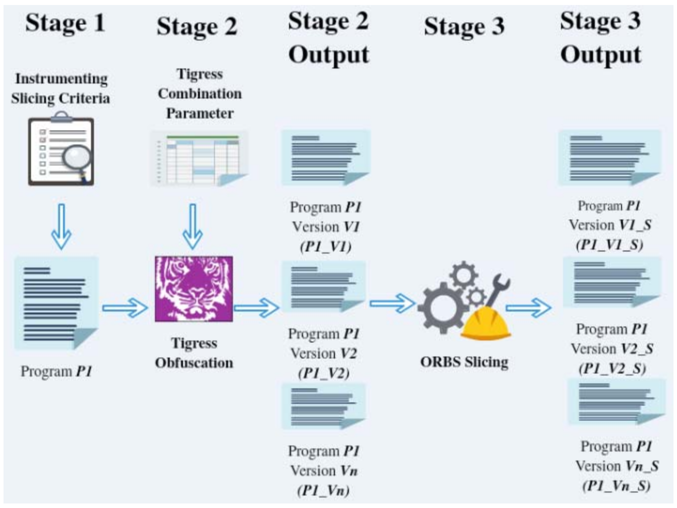
Stage 1
Stage 2
Stage 2 Output
Stage 3
Stage 3 Output
Instrumenting Slicing Criteria
Tigress Combination Parameter
Program P1
Tigress Obfuscation
Program P1 Version V1 (P1_V1)
Program P1 Version V2 (P1_V2)
Program P1 Version Vn (P1_Vn)
ORBS Slicing
Program P1 Version V1_S (P1_V1_S)
Program P1 Version V2_S (P1_V2_S)
Program P1 Version Vn_S (P1_Vn_S)

Fig. 1. Obfuscation resilience checker Toolchain.

out the parts related to the point of interest and reduce the amount of code that the attacker needs to analyse. We use ORBS to slice the variant, to reduce the number of lines that the attacker would need to study for the same points of interest as in the original program. These are marked as *P1_V1_S* to *P1_Vn_S*, where *n* is the number of variants generated in the obfuscation phase. Our conjecture is that the resilience of obfuscation and its quality is correlated to the amount of code that needs to be manually inspected by the attacker to understand the program. Therefore, the more successful ORBS is at reducing the program (the smaller the slice) the easier it would be for the attacker to read and understand the code.

Finally, the results are analysed for each different obfuscated variant of the program and compared to the original to determine which Tigress transformation performs better than others. The approach presented in this paper therefore is a framework that not only helps identify which of the obfuscated variants from a set is more resilient, but that can be later extended to identify which of the configuration parameters of the obfuscator will produce the best results.

IV. Experimental Setup

A. Program Selection

In order to conduct the experiment, we selected 8 open source C programs that are widely used in program slicing. The programs have different size, structure and functionality to ensure that we cover programs with different properties. Table I gives information about the program sizes in terms of Lines of Code (LoC), functionality and the slicing variable used in our experiments.

B. Transformation Selectors

The Tigress tool has a number of parameters: out of the several code transformation options, we selected *Virtualization*, *Control flow flattening* and *Function splitting* which are the most advanced offered by the tool.

TABLE I
Experimental subjects

Program	Functionality	Lines	Slice Criteria Variable(s)
P1	Month Calculator	45	`month_days, mnames`
P2	Delta Square Root	54	`i`
P3	Simplified Fibonacci	35	`n, s`
P4	Name number generator	173	`counter`
P5	Fibonacci (extended)	84	`n, result`
P6	Sequence generator	127	`n, result`
P7	Simple calculator	51	`summation`
P8	Resource allocator	40	`i`

Based on these transformations and their parameters, we were able to generate from 8 to 15 variants for each of the open-source programs included in the experiment. It is important to note that some combinations of parameter values do not produce valid configurations due to some conflicts among the obfuscator parameters; sometimes Tigress is able to detect this invalid configurations by stopping the build process and reporting errors, while in other cases the build process finishes but the code built might not be executable. The Tigress transformation parameters and their options used in this study are listed in Table II. It should be noted that the *seed* parameter that selects a random transformation has been set to 0 to exclude any randomness in code generation from our experiments as we could not track the correspondence between a particular seed value and the set of generated transformations. Moreover, we did not consider `switch`, `indirect` and `call` parameter values of the *Control-flow flattening* transformations, because of its similarity with the *Virtualization* transformation, and also because we noticed that often such combinations did not build a valid C file, due to the conflicts between these two types of transformations.

The three transformation parameters used could take a total of 20 values, resulting in almost 200 possible combinations. Some of these combinations worked for some programs and not for others, thus we were able to generate 48 combinations that worked across the various programs; we generated 8 to 15 obfuscated versions for each of the programs: this ultimately resulted in a total of 90 obfuscated variants overall for the 8 subject programs.

C. Orbs Configuration Selection

Finally, the relevant configurations for the ORBS slicers is given in Table III. We do not discuss the parameters in detail as ORBS is deterministic and we used the parameters consistently throughout the experiments. There are further ORBS parameters that could be set, but only the minimal compulsory set was used.

Each experiment subject provides multiple variables that produce output: an attacker might be interested in one or more of these variables. Column 4 of Table I shows the slicing criteria that we selected for each subject programs. One key point to note is that, in order for the ORBS tool to perform slicing, we had to instrument the source code with `ORBS`

TABLE II
TIGRESS PARAMETERS

Transformation and option name and description	Parameter	Description
Virtualization (VirtualizeDispatch) - Turn a function into a specialized interpreter.	`switch`	dispatch by while()switch(next)...
	`direct`	dispatch by direct threading
	`indirect`	dispatch by indirect threading
	`call`	dispatch by call threading
	`ifnest`	dispatch by nested if-statements
	`linear`	dispatch by searching a table using linear search
	`binary`	dispatch by searching a table using binary search
	`interpolation`	dispatch by searching a table using interpolation search
	`?`	Pick a random dispatch method(not used)
Control-flow Flattening (FlattenDispatch) Remove control flow from a function. (switch, indirect, call) generates error and * is randomly selected.	`switch`	dispatch by while(1) switch (next) blocks
	`goto`	dispatch by labl1: block1; goto block2;
	`indirect`	dispatch by goto* (jtab[next])
	`call`	each block is outlined into its own function
	`*`	select a dispatch method at random.(not used)
Function splitting (SplitKinds) - Split a function into smaller parts.	`top`	split the top-level list of statements into functions
	`block`	split a basic block (list of assignment and call statements) into two functions.
	`deep`	split out a nested control structure of at least height> 2 into its own function
	`recursive`	same as block, but calls to split functions are also allowed to be split out.
	`level`	split out a statement at a level specified by –SplitLevel.
	`inside`	split out a statement at the innermost nesting level.

TABLE III
ORBS CONFIGURATION PROPERTIES

Name of the property	Value
Slicing window size	4
Slicing direction	backward

keyword to capture the trajectory for the slicing criteria; the instrumentation process does not impact the functionality or semantics of the program.

V. Experimental Results

In this experiment the Toolchain has computed 90 variants of 8 programs to determine the *resilience* of the obfuscated code generated by Tigress. The Toolchain computes both the increased number of lines after obfuscation and the redundant code removed by ORBS: the actual lines of code left after ORBS processing indicate the *resilience* of the obfuscation. Figure 2 graphically represents the results of the increase in the program sizes after the transformation. We can see that depending on each transformation parameter, program size varies widely across different variants.

For example, there are programs with initial size of 50 LoC which have increased to over 2000 lines for a given set of parameters, whereas some have only increased to 500 for the same set of parameters. However, if it is possible for an attacker to use automated tools to remove the amount of inflation in lines and reverse it back to nearly the original size, the effectiveness of the obfuscation transformation can be considered questionable.

Our conjecture is that executing ORBS using slicing criteria for a program will determine the least number of lines that a MATE attacker needs to look at and understand in order to reverse engineer. Lines that ORBS is not able to remove

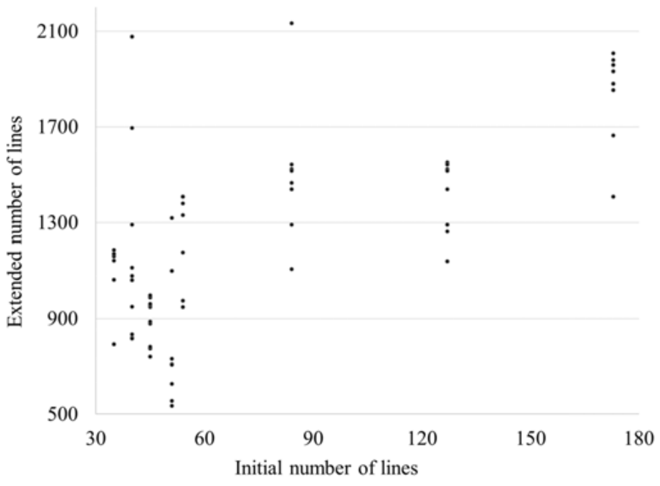

Fig. 2. Program expansion using Tigress obfuscation.

from each variant represent the least number of lines that are required to understand the behaviour of the particular point of interest.

Table IV shows how much of the original program can be removed by ORBS before applying any obfuscation. Column 2 reports the number of lines in the original program, column 3 reports the number of lines that were deleted, and columns 4 and 5 reporting the number of lines remaining and the percentage of lines removed, respectively. Column 4 therefore shows the least number of lines required to execute the program while producing the same results for the selected slicing criterion. From Table IV, it can be seen that ORBS was successful at removing a minimum of 45% of the program code (*P8*) leaving a potential attacker with 55% of the code to study. On the other hand, ORBS succeeded at removing 95%

of code (*P4*), in which case an attacker would be left with only 5% of code to analyse.

The graph in Figure 3 summarises 90 variants of 8 programs and represents the resilience of obfuscated code by removing redundant code using the developed tool-chain. The black line represents removal of redundant code using ORBS, while the grey line represents the *resilience* of obfuscated code, i.e. the percentage of obfuscated code that could not be removed by ORBS.

The *resilience* in Figure 3 reads from left to right; where left side of the graph represents higher *resilience* than of the right. The graph clearly shows obfuscated code for *P8_V27* (Program 8 - version 27), has much higher resilience than *P4_V16* (Program 4 - version 16). For example, initially *P8* had 40 LoC which was increased to 950 LoC and only 130 LoC were removed by ORBS. The Tigress combination `ifnest`, `goto` and `recursive` was used to generate this variant which provides 86% resilience.

Table V presents data for the highest resilience achieved by the variants of each program. Looking at the resilience (Table V) for each transformation, it is not evident that any particular set or combination parameters outperforms the others. Therefore, the current set of results seem to show that the structure of the program also significantly influences the resilience resultant from the application of particular parameters. For instance, *P1* initially had 45 lines of code, this was increased by 2216% lines of code with the Tigress parameter `interpolation goto recursive`, with the coding exhibiting a resilience of 60%. On the other hand, Tigress combination parameter `ifnest goto top` produced resilience of 67% with code inflation of 1951%. Therefore, we can argue that although combination `interpolation goto recursive` produced maximum increase in the number of lines, `ifnest goto top` still provides higher resilience from attack.

Although there is no clear trend, looking at the ranked results, the combination `ifnest goto deep` produced the highest resilience at 86% for *P8_V27* by removing only 14% of redundant code. We found that increased number of lines did not necessarily increases the *resilience* of obfuscated code. Future work will further look into details of correlations between applied transformation parameters, programming constructs (loops, branches etc.) and the identified resilience.

Furthermore, from Table V, it can be seen that the transformation for *P8* was highly resilient when compared to *P4*. The data for *P8* actually mimics the results published in Table IV. For program *P8*, ORBS obtained a reduction of only 45% when slicing the original program and variants of the same program also exhibits highest resilience at 86%. Similarly, for program *P4* whose variants have the lowest resilience, ORBS produced the smallest relative slice at 95% reduction of the program. Although the data for the best and worst performers map to each other, the mapping does not hold for the programs that lie in the middle of the range.

A. Threats to Validity

There are several threats to validity that we have identified. These include the selection of programs and whether they represent real-world systems. We also acknowledge that all possible combinations of Tigress parameters have not been explored in this paper. The combinations of parameter values for Tigress were randomly generated out of a set of valid values but all variants had to be compiled and run to check if they were working. In some cases this resulted in variations to be produced correctly for a certain set of programs but failing for others; making the comparison unbalanced. We expect to be able to address all such and additional threats to validity in a future extended empirical study.

VI. Related Work

There are a number of commercial and non-commercial tools like IDA, GDB, Ghidra, Radare2, OllyDbg, Valgrind, and the Angr framework that have been demonstrated as effective for the purpose of reverse engineering [19]. Static analysis and dynamic analysis are the key choice of program analysis that are used for reverse engineering [20]. Shimba is a tool where static information is extracted from bytecode and dynamic events are traced automatically from selected control flow objects [20]. In contrast, Udupa et al proposed a dynamic analysis approach to enhance the reverse engineering process [21]. In their experiment they have used artificial blocks of code into two different functions: only one is executed while the other is a decoy that can be spot with control-flow analysis.

Different metrics are used to measure various security requirements [22], and similarly code metrics have also been a common approach to measure obfuscation strength [23] or by calculating their potency [3]. Other approaches have been proposed to measure the attacker effort increased by obfuscation by means of controlled experiments with students [23], penetration testers [24] or public challenges [25], while other works tried to represent the attacker effort with modelling approaches based on Petri nets [26] [27], or an ontology of attacks and protections [28].

Yadegari et al. showed how to undo complex obfuscation techniques [29] by noticing that weak protections only inject bogus code with invariant behaviour that can be identified more easily. Liu et al. used resilience as optimisation function to search for the best obfuscation in 20 popular Javascript projects [30]. None of these techniques consider program slicing as an effective approach to reverse engineer. Sebastian et al. proposed to characterise the resilience of code obfuscation, transformations against automated symbolic execution [31]. Subsequently further work has also been proposed to improve this ability by using a combination of fine-grained bit-level taint analysis and architecture-aware constraint generations [32]. Scrinzi et al. extracted semantic information and behaviour of the execution for de-obfuscation [33]. They argued that there are some characteristics of the execution that are strictly correlated with the underlying logic of the code which are invariant after applying obfuscation. The search for the

TABLE IV
TARGET PROGRAMS AND LINES OF CODE REQUIRED TO EXECUTE

Program	Number of Lines	Lines deleted	Lines remaining	% of lines removed
P1	45	29	16	64%
P2	54	29	25	54%
P3	35	23	12	66%
P4	173	164	9	95%
P5	84	64	20	76%
P6	127	107	20	84%
P7	51	33	18	65%
P8	40	18	22	45%

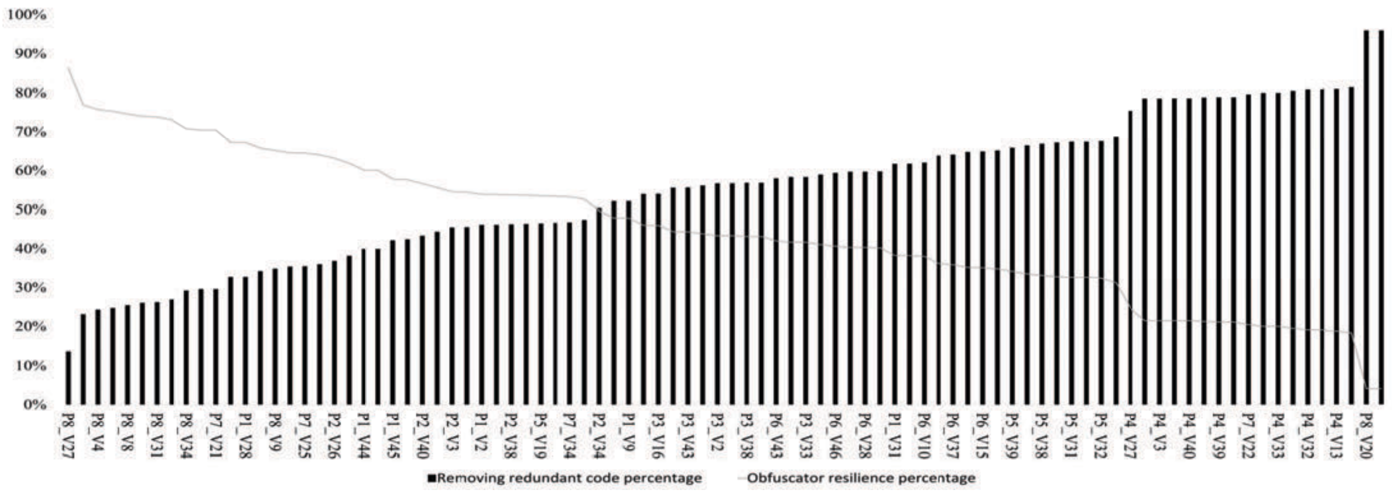

Fig. 3. Redundant code removed by ORBS to determine resilience of Tigress obfuscation.

TABLE V
HIGHEST RESILIENCE

Program	Tigress Combination	Resilience%
P1	`ifnest goto top`	67%
P2	`interpolation goto block`	66%
P3	`indirect goto deep`	46%
P4	`ifnest goto deep`	25%
P5	`call goto top`	54%
P6	`interpolation goto top`	42%
P7	`call goto top`	70%
P8	`ifnest goto deep`	86%

optimal obfuscation configuration has been done by Ceccato et al. [34] in case of software diversity but using static metrics. To the best of our knowledge we are the first to propose slice size as a measure of resilience: our approach mimics the use of debugging by actual attackers and aims to automate the resilience measurement.

VII. EXPERIMENTAL DATA

Full set of our experimental data and results available at http://syedislam.com/obfuscation.

VIII. SUMMARY

In this paper we introduced a novel approach to compute obfuscation *resilience* using program slicing. Our main goal was to determine which obfuscated variant of a program provides highest protection against MATE attacks. In general, larger obfuscated code may be perceived as harder to understand and reverse engineer, but we showed that this is not always the case, when the attacker uses program analysis tools for de-obfuscation and code size reduction. After running our program slicing tool, we assumed that the resulting slice size could be considered a measure of resilience of the obfuscated code. Our results confirm our conjecture that obfuscated variant size (LoC) is not a direct measure of quality of resilience as we found that larger size variants may be simplified by program slicing, thus showing low resilience. Future work will look at improving our metrics for several case studies, and using them to search for the optimal obfuscation parameters and configurations that maximise resilience and other code complexity metrics.

REFERENCES

[1] BSA, "Business Software Alliance BSA global software survey 2017," https://gss.bsa.org/, 2018, [Online; accessed 20-January-2019].

[2] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, "Guest Editors' Introduction: Software Protection," *IEEE Software*, vol. 28, no. 2, pp. 24–27, 2011. [Online]. Available: http://ieeexplore.ieee.org/document/5720710/

[3] C. Collberg and J. Nagra, *Surreptitious software : obfuscation, watermarking, and tamperproofing for software protection*. Addison-Wesley, 2010.

[4] P. Falcarin, R. Scandariato, and M. Baldi, "Remote trust with aspect-oriented programming," in *20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, vol. 1, April 2006, pp. 6 pp.–458.

[5] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, aug 2002. [Online]. Available: http://ieeexplore.ieee.org/document/1027797/

[6] R. I. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program," Sep. 24 1996, uS Patent 5,559,884.

[7] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in google play," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 222–235.

[8] J. M. Nardone, R. T. Mangold, J. L. Pfotenhauer, K. L. Shippy, D. W. Aucsmith, R. L. Maliszewski, and G. L. Graunke, "Tamper resistant methods and apparatus," Jan. 23 2001, uS Patent 6,178,509.

[9] J. Davis, "Protecting intellectual property in cyberspace," *IEEE Technology and Society Magazine*, vol. 17, no. 2, pp. 12–25, 1998. [Online]. Available: http://ieeexplore.ieee.org/document/682891/

[10] Weiser and M. David, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," 1979. [Online]. Available: https://dl.acm.org/citation.cfm?id=909356

[11] S. Hada, "Zero-knowledge and code obfuscation," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2000, pp. 443–457.

[12] Semdesigns.com, "Semantic Designs: Source Code Obfuscators," http://www.semdesigns.com/Products/Obfuscators/index.html?Home=Main, 2018, [Online; accessed 11-January-2018].

[13] Irdeto.com, "Irdeto Cloaked CA ships 500,000 units in first six months: Europe - Irdeto," https://www.irdeto.com/news/irdeto-cloaked-ca-ships-500-000-units-in-first-six-months-europe.htm, 2012, [Online; accessed 14-January-2018].

[14] Stunnix.com, "C/C++ Obfuscator," http://stunnix.com/prod/cxxo/, [Online; accessed 30-January-2018].

[15] C. Collberg, "Tigress obfuscator," http://tigress.cs.arizona.edu/, [Online; accessed 31-January-2019].

[16] D. Goswami and R. Mall, "Dynamic slicing of concurrent programs," in *International Conference on High-Performance Computing*. Springer, 2000, pp. 15–26.

[17] N. Sasirekha, A. E. Robert, and M. Hemalatha, "Program slicing techniques and its applications," *International Journal of Software Engineering & Applications (IJSEA)*, vol. 2, no. 3, 2011. [Online]. Available: http://www.airccse.org/journal/ijsea/papers/0711ijsea04.pdf

[18] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS: language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. New York, New York, USA: ACM Press, 2014, pp. 109–120. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2635868.2635893

[19] C. Taylor and C. Collberg, "A tool for teaching reverse engineering." in *ASE@ USENIX Security Symposium*, 2016. [Online]. Available: https://www.usenix.org/system/files/conference/ase16/ase16-paper-taylor.pdf

[20] T. Systä, *Static and dynamic reverse engineering techniques for Java software systems*. Tampere University Press, 2000.

[21] S. Udupa, S. Debray, and M. Madou, "Deobfuscation: Reverse Engineering Obfuscated Code," in *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE, 2005, pp. 45–54. [Online]. Available: http://ieeexplore.ieee.org/document/1566145/

[22] S. Islam and P. Falcarin, "Measuring security requirements for software security," in *2011 IEEE 10th International Conference on Cybernetic Intelligent Systems (CIS)*. IEEE, 2011, pp. 70–75.

[23] M. Ceccato, A. Capiluppi, P. Falcarin, and C. Boldyreff, "A large study on the effect of code obfuscation on the quality of java code," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1486–1524, dec 2015. [Online]. Available: http://link.springer.com/10.1007/s10664-014-9321-0

[24] M. Ceccato, P. Tonella, C. Basile, B. Coppens, B. De Sutter, P. Falcarin, and M. Torchiano, "How professional hackers understand protected code while performing attack tasks," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 154–164.

[25] M. Ceccato, P. Tonella, C. Basile, P. Falcarin, M. Torchiano, B. Coppens, and B. De Sutter, "Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge," *Empirical Software Engineering*, pp. 1–47, 2018.

[26] G. Zhang, P. Falcarin, E. Gomez-Martinez, S. Islam, C. Tartary, B. De Sutter, and J. d'Annoville, "Attack simulation based software protection assessment method," in *2016 International Conference On CyberSecurity And Protection Of Digital Services (Cyber Security)*, June 2016, pp. 1–8.

[27] Q. Su, F. He, N. Wu, and Z. Lin, "A method for construction of software protection technology application sequence based on petri net with inhibitor arcs," *IEEE Access*, vol. 6, pp. 11 988–12 000, 2018.

[28] C. Basile, D. Canavese, L. Regano, P. Falcarin, and B. De Sutter, "A meta-model for software protections and reverse engineering attacks," *Journal of Systems and Software*, vol. 150, pp. 3–21, 2019.

[29] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *IEEE Symposium Security and Privacy*, 2015, pp. 674–691.

[30] H. Liu, C. Sun, Z. Su, Y. Jiang, M. Gu, and J. Sun, "Stochastic optimization of program obfuscation," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 221–231.

[31] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications - ACSAC '16*. New York, New York, USA: ACM Press, 2016, pp. 189–200. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2991079.2991114

[32] S. Banescu, C. Collberg, and A. Pretschner, "Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning," *26th USENIX Security Symposium*, 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/banescu

[33] F. Scrinzi, "Behavioral analysis of obfuscated code," Master's thesis, University of Twente, 2015.

[34] M. Ceccato, P. Falcarin, A. Cabutto, Y. W. Frezghi, and C.-A. Staicu, "Search based clustering for protecting software with diversified updates," in *International Symposium on Search Based Software Engineering*. Springer, 2016, pp. 159–175.