Program Obfuscation: A Quantitative Approach

Bertrand Anckaert, Matias Madou,
Bjorn De Sutter, Bruno De Bus
and Koen De Bosschere
Ghent University

Bart Preneel
Katholieke Universiteit Leuven

ABSTRACT

Despite the recent advances in the theory underlying obfuscation, there still is a need to evaluate the quality of practical obfuscating transformations more quickly and easily. This paper presents the first steps toward a comprehensive evaluation suite consisting of a number of deobfuscating transformations and complexity metrics that can be readily applied on existing and future transformations in the domain of binary obfuscation. In particular, a framework based on software complexity metrics measuring four program properties: code, control flow, data and data flow is suggested. A number of well-known obfuscating and deobfuscating transformations are evaluated based upon their impact on a set of complexity metrics. This enables us to quantitatively evaluate the potency of the (de)obfuscating transformations.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*protection mechanisms*; K.4.4 [**Computing Milieux**]: Electronic Commerce—*security*

General Terms

Measurement, Security

Keywords

Program Obfuscation, Quantification, Metrics

1. INTRODUCTION

The goal of program obfuscation is to delay program understanding. Barak et al. [5] have shown that no omnipotent obfuscation exists, while Appel [1] has shown that deobfuscation is NP-easy. Still many transformations have been proposed that do, intuitively, make the understanding of a program harder, and in some cases even impossible [23].

Obfuscating transformations aim to transform the program into a semantically equivalent program which is much harder to understand for an attacker. The most popular transformations are the insertion of opaque predicates [12], the flattening of the control flow graph [34], the insertion of self-modifying code [3] and corrupting the disassembly [22]. Obfuscation researchers typically insert *complex* code or data structures into programs, thus supposedly making the programs harder to analyze. Very few papers, however, discuss the resilience of their proposed transformations or even the complexity of the obfuscated code.

In addition to the popular obfuscating transformations, recent papers discuss how these transformations can be (partially) undone. For example, the default control flow flattening was broken by applying well known static and dynamic analyses [33]. However, the reverse transformation is not perfect: breaking the flattening algorithm leaves a few additional edges in the program. In this paper we evaluate how obfuscating and deobfuscating transformations affect the understanding of the program. To this end, we apply software complexity metrics to evaluate the quality of the applied obfuscating and deobfuscating transformations.

This paper takes the first step in creating a unified suite for the evaluation and comparison of obfuscating and deobfuscating transformations. Its major contribution is the proposal of a framework for comparing and evaluating obfuscating transformations by means of a unified set of quantitative metrics and a corresponding taxonomy measuring four program properties; *code*, *control flow*, *data* and *data flow*. These metrics as such do not evaluate the resilience of the transformations. Although there sometimes exists a deobfuscating transformation, it is unknown if it is applicable to all obfuscated programs due to requirements of the deobfuscating transformation or the additional protection by tamper- resistance transformations. For example, breaking control flow flattening [33] might require a control flow graph which can be hidden through additional transformations. As such, deobfuscating transformations can provide a lower bound to how well the complexity can be reduced when a specified set of obfuscating transformations has been applied: future deobfuscating transformations can reduce the complexity even further, while additional protecting transformations can render them less powerful.

This paper focuses on the obfuscation of binary executables, containing no symbolic information. We limit ourselves to automated transformations, that can be applied by tools such as compilers or binary rewriters. Finally, we focus on techniques that protect against an attacker who knows how to run a program. This excludes techniques such as total program encryption which offers no protection against legitimate owners of software, who possess all the keys.

This paper is organized as follows. Section 2 discusses the need of an evaluation suite and motivates why the evaluation in cryptography cannot be used. Section 3 presents concrete metrics for evaluating program obfuscating transformations, which are evaluated experimentally in Section 4. Conclusions are drawn in Section 5.

2. AN EVALUATION SUITE

Our ultimate goal is an evaluation suite that allows researchers to compare the complexity of their applied obfuscating transformations. Similar evaluation suites exist in

QoP'07, October 29, 2007, Alexandria, Virginia, USA.

other domains, e.g., the SPEC benchmark suite allows comparing compiler optimization techniques, computer architectures and processor implementations or the StirMark [30] suite which tests the robustness of image watermarking algorithms. Our evaluation suite will contain deobfuscating transformations and complexity metrics.

Clearly, such an evaluation suite can never replace a thorough theoretical evaluation of the applied obfuscating transformations. Unfortunately, theoretical advances in this field are, while definitely worthwhile, slow. On one side of the spectrum, Barak et al. [5] have shown that perfect obfuscation is impossible. On the other side of the spectrum, provable obfuscations for complex access control functionalities in the random oracle model [23] and for hash functions under non-standard number-theoretic assumptions [8] have been constructed.

In between these two extremes, there is a large gap for which little theoretical background is available. Many transformations have been proposed that do, intuitively, make the understanding of a program harder, while not having provable properties. One high-profile example is the Internet telephony software by Skype [6], which has allegedly been thoroughly obfuscated. The goal is obvious, to hide its business model from rival companies to prevent competing products from offering the same level of service or even from inter operating with the existing installed base of Skype.

This lack of theoretical background contrasts with the domain of cryptography, where the notions used include semantic security (which is the computational version of perfect secrecy as achieved by the one-time pad) and indistinguishability of ciphertexts [15]. An encryption scheme is considered to be secure in the latter sense if it infeasible for a computationally bounded attacker who is given a ciphertext C, and two plaintexts P_1 and P_2, to determine with probability essentially better than $1/2$ which plaintext was encrypted. With a one-time pad, an attacker can always by luck guess the right key and decrypt, but of course this does not endanger the security of encryption, just like the existence of random deobfuscating transformations does not endanger the strength of obfuscating transformations.

We could consider a similar distinguishing test to evaluate obfuscating transformations. In such a test, an attacker would be given any two plain programs (with the same functionality) and one obfuscated program. The obfuscating transformation would be considered strong enough if enough time would be needed to determine which of the two plain programs corresponds to the obfuscated one.

Unfortunately, this is not possible. Fundamentally, obfuscation differs from encryption in at least three aspects. First, there is no need for the existence of an inverse transformation. Secondly, an attacker is not necessarily looking for the original program, and an attack can be successful without finding the original program. And thirdly, ciphertext should be meaningless without the key while an obfuscated program should still execute within reasonable performance constraints without additional information.

Even if it were possible to create a transformation which passes the distinguishing test, it would not be very useful within the context of obfuscation. To show this, we discuss an example where this test is passed, but the goals of obfuscation are not met. Suppose that we have invented a new, superior algorithm P_1, and we have also invented a second algorithm, P_2, that is, in all respects, as efficient as P_1. We then apply a transformation on either P_1 or P_2 resulting in an equivalently complex version C. Suppose that it is impossible to tell with a probability significantly higher than $1/2$ whether C is derived from P_1 or P_2, then the test is passed. However, if C is no more complex than either P_1 or P_2, this inability is of no concern to an attacker. Indeed, an attacker can simply analyse C to understand the inner workings, with all the good properties of both P_1 and P_2.

Clearly, an obfuscation metric should indicate whether or not an obfuscated program is more complex than the original with respect to program understanding.

Similar to how the SPEC benchmark suite has evolved over time because of changes in computer architecture, and compiler optimizations, we believe an obfuscation evaluation suite should evolve over time. For example, when new types of code constructs are inserted in programs as a way to obfuscate them, the deobfuscating transformations should be adapted to the existence of those constructs. This can be compared to the evolution of StirMark [30] and to the inclusion of ever more complex programs in performance benchmarks to avoid benchmark-specific compiler optimizations. Furthermore, just like the SPEC performance benchmark suite has been used with multiple metrics, such as execution time, instructions per cycle, operations per milliwatt, ... we expect an obfuscation evaluation suite to be used with multiple metrics.

While no generally applicable evaluation metrics have been proposed for use in the research of binary obfuscation, solutions have been proposed for other domains. On the one hand, Collberg [11] proposed the use of software complexity measures to guide the Java-obfuscator tool to choose the best sequence of obfuscating transformations [19]. On the other hand, specific evaluation metrics have been proposed by authors presenting new obfuscating transformations, such as Linn's *confusion factor* [22], but these lack general applicability. Other metrics require a decompilation step before the actual measurement can happen. Finally, metrics such as the *depth of parse trees* to measure the strength of source code obfuscation [16] are not applicable in the domain of binary obfuscation.

3. CONCRETE METRICS

In this section, we propose to apply concrete Software Complexity Metrics (SCMs) on four fundamental program properties: *instructions*, *control flow*, *data flow* and *data*. This proposal is based on the following observations.

Most importantly, we believe these four properties, although not always fully orthogonal, are as close as possible to a four-dimensional space in which concrete, mostly independent, SCMs can be applied on all four axes. Let us consider a concrete problem to illustrate this: the sorting of credit card numbers. If the instructions implementing the sort are not recognizable, surely the sorting algorithm itself is not recognizable. Knowing all instructions, however, does not necessarily imply that we know all possible execution sequences of those instructions, i.e., the control flow. To understand the sorting algorithm, one certainly needs to know its control flow. But even if we understand that, it may not be clear which data is being moved when, or why. So without having at least a partial understanding of the data flow, the algorithm will not be understood. Finally, even if we know that an array of numbers is being permuted, as happens with in-place sorting, and on what basis the per-

mutation takes place, we do not necessarily know what the actual data is. It might be, e.g., that the credit card numbers have been hashed or encrypted.

Furthermore, we believe that these axes roughly correspond to four phases in program obfuscation, which will ease the positioning of proposed obfuscating transformations on the axes. First, a developer might want to change his algorithms to hide which data is being computed. This type of transformation happens at the source level, as it requires domain-specific knowledge of the application. Secondly, an obfuscator will try to hide which concrete operations are being executed on which data. This type of transformation can also often be achieved by transforming source code. Next, the obfuscator of a program will try to hide the order in which the (obfuscated) operations of a program will be executed. This type of transformation will typically be performed at the intermediate code level used by compilers. Finally, the obfuscator might try to hide the executed operations themselves. This will most likely be implemented in a post-pass tool, such as a link-time rewriter.

Likewise, an attacker will attack a program in the reverse order. First he will try to see which instructions are being executed. Then he will try to determine the control flow, and finally he will try to track the data on which the program operates to understand the implemented algorithms or to extract sensitive data from the program.

3.1 Instructions

Obviously, the static and dynamic sizes of the set of instructions used in a program can be used as a simple SCM. The reasoning behind this SCM is that programs become more complex when more different instructions are (potentially) executed [17]. Note that dynamic coverage analysis will not take into account unreachable code added to a program. Such analysis will hence make sure that adding unreachable code is not considered useful.

Related complexity metrics on instruction complexity are possible too: higher weights could be given to more complex instructions, for a wide range of *instruction complexity* metrics. For example, rare instructions, for which an attacker might need to consult the reference manual in order to understand them, might be considered more complex [13]. Alternatively, the weight might depend on the number of non-immediate operands of instructions. The latter can either be statically, meaning the *number of source (register) operands* of an instruction, or dynamically. In the latter case, the number of *different numerical values of the operands* can even be taken into account.

3.2 Control Flow

The order in which the instructions are executed is equally important. This order is reflected in the control flow. In general, the more complicated the control flow, the more complex a program. For source code, several control flow SCMs have been proposed.

McCabe [26] proposed the use of the *cyclomatic number* $e-n+2p$, in which e is the number of edges, n the number of nodes, and p the number of connected components. As such it is an indication of the number of decision points in the program and represents the number of linearly independent paths through the code. Woodward [36] proposes to use the *knot count*. This measures the unstructuredness of a CFG, as it measures the number of crossings of control flow arrows in a graph. The more crossings there are, the more difficult it is to follow the execution of a program. To make the knot count independent of the layout of the CFG, all code is represented in a vertical layout, in the order in which it appears in the source code, and all edges are drawn on the same side of the sequence. Harrison [18] proposed using a combination of program size and control flow, in which nodes (basic blocks) are weighted with their *nesting depth*. Clearly, many SCMs can be constructed. They have in common that adding edges to a graph increases the measured complexity.

3.3 Data Flow

Data flow relates to the production and consumption of numerical values by instructions. In executable programs, this usually involves the propagation of (untyped) values through registers and through memory locations.

SCMs based on static program representations can include the size of sets of live values, the size of points-to sets, numbers of def-use pairs [32], the size of program slices, etc. The reasoning behind such SCMs is that the larger the sets are, the more information will need to be remembered by an attacker to understand a program. A well known, concrete example of such SCMs is the *fan-in/fan-out* [20] of procedures, being the number of formal parameters and the number of global data structures read/written by procedures.

For dynamic program representations consisting of program traces, similar properties can be considered. Distance-based SCMs that relate to the distance (in number of executed cycles) between productions and consumptions of values can be considered. The reasoning there is that longer distances will force the attacker to look at the traces with wider windows, i.e., to use larger working sets when trying to extract knowledge from the traces.

Finally, there also exist SCMs on data structures used in a program. For example, Munson and Khoshgoftaar [27] attribute a constant complexity to scalar variables, while the complexity of an array increases with the number of its dimensions and with the complexity of the element type, and the complexity of a record increases with the number and complexity of its fields. At first sight, such SCMs may seem useless because binary programs only operate on data in memory locations. Deobfuscating transformations exist, however, that decompile programs to the extent that stack-allocated, statically allocated, or even dynamically allocated heap data are given meaningful semantics. Balakrishnan and Reps [4], e.g., are able to differentiate between spilled data and procedure parameters in stack-allocated data.

3.4 Data

Concrete SCMs on data, i.e., on values occurring during a program's execution or in its statically allocated data, are much harder to develop than SCMs on the three already discussed program properties. This results from the fact that obfuscating transformations that try to hide the meaning of actual values occurring in a program, most often rely on domain-specific knowledge and about specific, high-level semantic knowledge about the algorithms being obfuscated. Furthermore, obfuscating transformations such as the ones relying on transforming data from one field to a second, isomorphic field (e.g., homomorphic encryption [29]) are applied manually at the source code level. As such, these transformations are out of scope for this paper.

4. EXPERIMENTAL EVALUATION

For our quantitative approach, we have adopted a number of SCMs to the domain of binary executables. We then

evaluated the impact of existing obfuscating transformations on these SCMs. Furthermore, we have evaluated how well existing deobfuscating transformations succeed in undoing the added complexity. These results have been obtained from the C programs of the SPECint 2000 benchmark suite.

4.1 Evaluated Complexity Metrics

The SCMs we used reside on the axes of code and control flow. First, we measured the static instruction count, i.e., the number of instructions in a CFG. Next, we measured the cyclomatic number and the knot count, see Section 3.2.

One of the most important requirements for SCMs in this domain is that they can be evaluated no matter what level of obfuscation has been applied to the binaries. We cannot rely on external information about the binary such as source code, heuristics about the used compiler or linker, ... Therefore, all metrics are computed using only information collected about the dynamic execution. Constructing the control flow graph containing only executed code is more precise than the static control flow graph. There is no uncertainty about the code in the former graph while the latter graph may be hard to construct in the first place.

To obtain these numbers, we constructed CFGs from dynamic traces collected with Diota [25], a dynamic instrumentation tool for the x86 architecture. The reconstructed graphs contain only basic blocks that were executed during the coverage analysis, together with edges along which control was transferred during the analysis. As such, the reconstructed graphs are non-conservative, and much smaller than the static, conservative graphs used to obfuscate. Consequently, they have a lower observed complexity. In practical attacks, we believe it is this sort of graph that is constructed and studied by attackers.

To get to a fair comparison, the same coverage analysis and graph reconstruction was applied on the original, obfuscated, and deobfuscated programs.

As far as we know, no techniques like ours have previously been developed to derive SCMs from traces. We therefore enter a new domain in the area of SCMs that we believe to be relevant in the context of software protection.

4.2 Evaluated Obfuscation Techniques

We have implemented some of the more popular existing structural obfuscating transformations for the x86 architecture by means of the retargetable link-time binary rewriting framework Diablo [7].

Control Flow Flattening.

Control flow flattening, as described by Wang [34] and Chow [9], tries to obscure the original control flow of a program by ensuring that one basic block, the *redirect* block, is the sole, common predecessor of all basic blocks in a procedure, as well as their sole, common successor. To guide control flow at run time, a switch table and a variable are inserted and the redirect block will redirect the control flow based upon the entry of the switch table identified by the variable. Flattening takes a central part in an industrial obfuscation tool by Cloakware Inc.[9]. An attack against this technique is discussed in [33].

In our implementation, each function is flattened individually, so each function has exactly one redirect block, and hence one switch table. A new entry basic block is added to the function to ensure that control flows from its new entry point to its original entry point over the redirect block.

Static Disassembly Thwarting.

Linn and Debray [22] insert branch functions and data into a program's code to thwart its static disassembly. In particular, they try to thwart the advanced disassembly heuristics discussed by Schwarz et al. [31]. Branch functions are functions that do not return to the caller; instead control is transferred to a different address computed from the return address on the stack and offset passed to the branch function. Furthermore, where possible, data is inserted in the code to try to get a linear disassembler out of alignment. In our implementation, a branch function is inserted and all unconditional jumps are transformed into calls to it. Secondly, all conditional jumps are inverted, which again introduces unconditional jumps. Finally, junk data is inserted after these newly-created call instructions as well.

Binary Opaque Predicates.

Opaque predicates [12, 28] have a property that is known at obfuscation time, but which is hard to discover afterwards. By using them for guiding control flow at run time, the control flow of a program can be cluttered with unrealizable paths. In the case of the binary opaque predicates that we implemented for this research, this property is the fact that they always evaluate to `true` ($\forall x \in \mathbb{Z},\ 2|x + x$ for example), or always evaluate to `false` ($\forall x \in \mathbb{Z},\ x^2 < 0$ for example). The opaque predicates we inserted are taken from Arboit [2]. To avoid simple elemination, link-time liveness analysis and constant propagation are used to ensure that the inserted predicate computations do not change the original program behavior, and to ensure that the inputs of the predicate computations are not constant values.

4.3 Evaluated Deobfuscation Techniques

In many recent papers, the authors explicitly or implicitly assume that either no automated program analysis tools will be used, or that only static analysis is used. This assumption is reflected in the titles of some publications. For example, Wang's technical report "Software tamper resistance: Obstructing static analysis of programs" [35], and "Obfuscation of executable code to improve resistance to static disassembly" by Linn *et al.* [22]. Consequently, the authors typically do not try to defend against the use of non-conservative, (partially) automated, dynamic analyses.

In general, conservatism means reporting weaker properties than the ones that may actually be true. This guarantees soundness, but often at the cost of information that is too imprecise to be useful. Attackers know this, and hence they rely on non-conservative analyses. In practice, advanced debuggers, such as SoftICE and IDA Pro are already commonly used. These tools allow for the incorporation of dynamically obtained information to form a view of the program. By contrast with conservative, static analyses, the information obtained dynamically is very precise, as it describes exactly what has happened during the execution. As this information only describes what has happened during a limited number of executions, it is not guaranteed to be sound information for all possible executions. For that reason, attackers will often be conservative only under realistic assumptions. They might assume, for example, that the calling conventions will be respected or that instructions will not overlap. In the end, attackers choose to use non-conservative analyses, either dynamic or static, whenever it suits them, or whenever they need to.

We implemented two existing deobfuscating transformations to attack control flow flattening and static disassembly thwarting. These attacks are based on an existing hybrid

static-dynamic attack [24] on watermarks based on branch functions [10]. A similar attack can be used to remove calls to a branch function which were inserted to thwart the static disassembly of a program. The static part of the attack consists of searching for the calls to the branch function in the binary image of the program. In our attack, we used the advanced static disassembly method introduced by Kruegel et al.[21]. The dynamic part consists of subsequently executing the branch function under the control of a debugger, thus extracting the correspondence between the branch function's callers and its branch targets. As we simply execute the branch function on its inputs obtained from the static analysis to detect this correspondence, the obfuscation by the branch function cannot be improved by using perfect hash tables or any other type of computation. Thus, this deobfuscating transformation does not depend on the "strength" of the branch function computations. Finally, each call to the branch function is replaced by an unconditional jump to the correct target.

A similar attack can be mounted against a binary obfuscated with control flow flattening [33]. The static part identifies the switch block and the jump-instructions to this block. The dynamic part then consists of executing the last instructions of the jump block to figure out which entry in the switch-table is used to transfer control. Finally, the jumps towards the switch are transformed into conditional and unconditional jumps directly to their target.

The deobfuscating transformations were limited to replacing part of the obfuscating instructions by simpler, but equally long instructions such as no-ops. Thus, not all instructions inserted during obfuscation are eliminated by the deobfuscating transformation. Also, the basic blocks constituting the program are not reordered during deobfuscation because the deobfuscation transformation does not construct the whole-program control flow graph. Consequently, the numbers we present in Section 4.4 are upper bounds that can still be lowered if one is only interested in building CFGs suited for program understanding, but not in generating working binaries out of them. Finally, we note that we are aware of one attack based on abstract interpretation which describes a group of easily breakable opaque predicates [14]. We did not find a general attack on opaque predicates in literature.

4.4 Impact of the Obfuscation Techniques

The static CFG contains unreachable code as well as reachable code. Of the reachable code, a large fraction is not executed during our tracing. Consequently, the numbers obtained from the dynamically constructed CFG are lower than those obtained from the static CFG. This is particularly so for the control flow metrics, because the dynamic version does not include the conservative estimate of indirect control flow transfers and the corresponding edges.

As can be seen in Table 1, none of the obfuscation techniques have a decreasing effect on the SCMs. For these benchmarks and with respect to the set of considered SCMs, the transformation based on opaque predicates is less potent than control flow flattening and static disassembly thwarting. Note that we consider a transformation T less potent than a transformation U with respect to a set of benchmarks B and a set of metrics M, if for every benchmark $b \in B$, for every metric $m \in M$, $m(T(b)) < m(U(b))$.

Furthermore, it can be noted that this transformation does not affect the cyclomatic number or the knot count. This is the result of computing the SCMs on the part of the CFG that was actually executed during a particular run. Therefore, the unrealizable paths do not appear in the metrics. As such, for simple opaque predicates, one basic block and one edge is added and the net impact on the cyclomatic number is zero. The impact on the knot count is also zero because the introduced edges do not cross any existing edges. We would like to note that this result does not imply that opaque predicates are not useful: we have only evaluated a straightforward transformation based upon this concept. Opaque predicates are a valuable primitive that have applications in many more domains.

Static disassembly thwarting is more powerful with respect to the cyclomatic number, while flattening is more powerful with respect to the instruction count and knot count. As a result, static disassembly thwarting and flattening cannot be ordered with respect to the *set* of SCMs considered. We thus get the following partial ordering:

$$opaque < \begin{pmatrix} flatten \\ static\ disassembly\ thwarting \end{pmatrix}$$

The cyclomatic number increases more through flattening than through static disassembly thwarting. This can be explained because each basic block is considered for flattening, while static disassembly thwarting only takes basic blocks ending with a jump into account. The knot count, on the other hand, is less after flattening than after static disassembly thwarting. The latter obfuscation technique uses a single redirection function, which causes many control transfers from all around the program to a single location. This results in many crossings. Flattening, on the other hand, has a switch block in each function limiting the new knots to crossing edges within the function itself.

In distinct cases, it is possible to compute the complexity of each obfuscated function based on the number of basic blocks b in the original function. For example, the cyclomatic number (CN) and knot count (KC) are for each flattened function given by: the $CN \cong b$ and $KC \cong b(b-1)/2$.

4.5 Impact of the Deobfuscation Techniques

Rows three and five of Table 1 illustrate how successful the deobfuscation techniques described in Section 4.3 are at eliminating the introduced complexity. Both deobfuscating transformations reduce the control flow metrics close to the complexity of the original program. Furthermore, the deobfuscation of flattening with respect to the cyclomatic number is perfect.

On the other hand, there is no significant reduction in the instruction count. As discussed earlier, the deobfuscating transformations replace part of the instructions inserted by obfuscation by simpler, but equally long instructions such as a sequence of no-ops. If required, an attacker can filter out these useless instructions when inspecting the program for the purpose of understanding.

Clearly, the obfuscating transformations increase the complexity of the program while the attacks lower the observed complexity to little more than the complexity observed on the original programs, the residue complexity.

5. CONCLUSIONS

We discussed why cryptographic tests are not suited to evaluate program obfuscation techniques, and instead proposed a quantitative approach based on software complexity metrics. Concrete metrics were presented for the program

Benchmarks		gzip	vpr	gcc	mcf	crafty	parser	perlbmk	gap	vortex	bzip2	twolf	% Increase
(a) Instruction Count													
Original		5,955	13,910	129,017	5,569	31,094	22,404	45,410	20,102	64,048	7,090	24,386	
Linn	obf	6,872	15,735	153,903	6,331	36,459	26,329	54,752	23,482	70,574	8,136	28,075	16.71
	deobf	6,395	14,763	141,028	5,973	33,394	24,579	49,076	21,479	67,088	7,584	25,999	7.69
Flatten	obf	12,869	24,363	213,383	12,409	53,841	50,113	78,935	38,876	122,314	15,645	50,750	82.53
	deobf	11,247	21,744	193,624	10,687	49,154	44,204	70,600	34,406	106,460	13,649	44,845	62.78
Opaque	obf	6,652	15,360	142,103	6,242	36,038	24,823	48,776	22,447	69,862	4,617	27,510	9.61
(b) Cyclomatic Number													
Original		373	907	13,892	279	2,584	2,530	4,201	2,450	3,640	425	1,818	
Linn	obf	481	1,139	16,880	370	3,149	2,908	5,307	2,902	4,697	578	2,297	22.99
	deobf	374	914	14,026	280	2,594	2,546	4,251	2,469	3,688	426	1,825	0.89
Flatten	obf	1,069	1,992	22,079	992	4,573	4,806	7,827	4,325	10,795	1,333	4,504	94.25
	deobf	373	907	13,892	279	2,584	2,530	4,201	2,450	3,640	425	1,818	0.00
Opaque	obf	373	907	13,892	279	2,584	2,530	4,201	2,450	3,640	425	1,818	0.00
(c) Knot Count													
Original		32,872	255,040	20,641,152	25,312	673,238	545,958	3,882,824	2,109,978	9,738,816	37,828	841,126	
Linn	obf	87,544	603,364	63,002,032	62,318	1,918,236	1,409,816	11,409,782	3,974,202	20,067,392	110,920	1,946,986	169.68
	deobf	33,766	266,356	21,936,152	25,986	694,522	587,928	4,186,978	2,172,332	10,144,128	39,150	859,756	5.58
Flatten	obf	77,328	315,712	22,331,272	56,480	1,335,528	973,822	4,168,312	2,285,332	10,586,828	148,254	1,177,930	12.05
	deobf	35,948	260,720	20,689,076	27,976	688,930	563,418	3,899,462	2,124,090	9,778,766	40,988	855,886	0.47
Opaque	obf	32,872	255,040	20,641,198	25,312	673,238	545,958	3,882,824	2,109,978	9,738,816	37,828	841,126	0.00

Table 1: (a) Instruction count, (b) cyclomatic number and (c) knot count computed on the machine code of the C-programs of the SPECint 2000 benchmark suite.

properties of code, control flow, data flow and data. Finally, these metrics have been applied to existing obfuscation techniques and existing attacks and we have demonstrated that our quantitative approach is suited to evaluate and compare the resulting complexity from obfuscation techniques.

6. REFERENCES

[1] A. Appel. Deobfuscation is in np, August 2002.

[2] G. Arboit. A method for watermarking java programs via opaque predicates. In *ICECR-5*, 2002.

[3] D. Aucsmith. Tamper resistant software: an implementation. *Information Hiding, Lecture Notes in Computer Science*, 1174:317–333, 1996.

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. Int. Conf. on Compiler Construction*, pages 5–23, 2004.

[5] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Advances in cryptology, LNCS*, 2139:1–18, 2001.

[6] P. Biondi and F. Desclaux. Silver needle in the skype. In *BlackHat Europe*, 2006.

[7] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. In *Proc. LCTES*, pages 211–220, 2004.

[8] R. Canetti. Towards realizing random oracles: Hash functions that hide all partial information. *CRYPTO 1997*.

[9] Cloakware Corp: S. Chow, H. Johnson, and Y. Gu. Tamper Resistant Software - Control Flow Encoding, Patent US 6,779,114, Filed 1999, Granted 2004.

[10] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Proc. PLDI*, 2004.

[11] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, 1997.

[12] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Conf. POPL*, pages 184–196, 1998.

[13] C. Cook. Information theory metric for assembly language. *Software Engineering Strategies*, pages 52–60, 1993.

[14] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. Opaque predicates detection by abstract interpretation. *Algebraic Methodology and Software Technology*, LNCS 4019:81–95, 2006.

[15] O. Goldreich. The foundations of cryptography. 2004.

[16] H. Goto, M. Mambo, K. Matsumura, and H. Shizuya. An approach to the objective and quantitative evaluation of tamper-resistant software. *ISW, LNCS*, 1975:82–96, 2000.

[17] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[18] W. Harrison and K. Magel. A complexity measure based on nesting level. *SIGPLAN Not.*, 16(3):63–74, 1981.

[19] K. Heffner and C. Collberg. The obfuscation executive. *Information Security Conference (ISC04)*, 2004.

[20] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 9 1981.

[21] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. of the 13the USENIX Security Symposium*, pages 255–270, 2004.

[22] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. *Proc. CCS 03*.

[23] B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *EUROCRYPT*, 2004.

[24] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proc. DRM 2005*.

[25] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, september 2002.

[26] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[27] J. Munson and T. Khoshgoftaar. Measurement of data structure complexity. *J. Syst. Softw.*, 1993.

[28] G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks, 2004.

[29] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. *LNCS*, 1592:223–238, 1999.

[30] F. Petitcolas, R. Anderson, and M. Kuhn. Attacks on copyright marking systems. In *Information Hiding*, pages 218–238, 1998. citeseer.nj.nec.com/petitcolas98attacks.html.

[31] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proc. WCRE 2002*.

[32] K. Tai. A program complexity metric based on data flow information in control graphs. In *Proc. ICSE*, pages 239–248, 1984.

[33] S. Udupa, S. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proc. WCRE 2005*.

[34] C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, Department of Computer Science, University of Virginia, October 2000.

[35] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 12 2000.

[36] M. Woodward, M. Hennel, and D. Hedley. A measure of control flow complexity in program text. *IEEE TSE*, 1979.