# Obfuscation through simplicity

Muhammad Rizwan Asghar Steven D. Galbraith Giovanni Russello

July 19, 2016

**Abstract**

The paper is about obfuscating control flow of programs against static and dynamic attacks, such as symbolic or concolic tools. We present a novel approach that converts branches of a certain type in code into single program segments with no branch statements. Consequently, running an automated tool such as a symbolic execution tool or any algorithm to compute a CFG will not reveal the branch instruction. Our broader theme is that "security from simplicity" may be a good approach to obfuscation, rather than "security from complexity".

## 1 Introduction

The problem of obfuscating computer programs remains a difficult challenge, for both applied and theoretical cryptography. Due to the large number of potential applications, it is an important problem to develop both theoretical and practical tools.

On the applied side, there is a large literature documenting an "arms race" of obfuscation proposals followed by attacks (for a recent survey see Schrittwieser et al [23]). The current most successful attacks seem to be automated de-obfuscation tools that dynamically analyse the code and compute Control Flow Graphs (CFGs) or other information.

On the theoretical side, there is also a conflict between the possible and the impossible. Barak et al [6] have shown that a natural strong notion of obfuscation (VBB) is impossible (in the sense that there are classes of programs that cannot be obfuscated). On the other hand, there is a large literature giving techniques and strong theoretical security arguments for the obfuscation of certain classes of functions (such as "evasive functions"). More recently, multilinear maps and constructions of indistinguishability Obfuscation (iO) have been proposed, though the state of this research area is currently unclear due to major cryptanalysis of multilinear maps.

Two recent surveys give an overview of the situation. Barak (ACM 2016) [5] surveys the theoretical constructions of iO and explains the power of the approach, while also mentioning concerns that this area is in danger of becoming a form of "post-modern crypto". Collberg (EUROCRYPT 2016) [15] emphasises the major challenges for practical obfuscation, and stresses that *"Finding the combination of primitive transformations and spatial and temporal diversity strategies that achieve the highest level of protection while staying within strict performance bounds is an unsolved engineering problem."*

However, it is very interesting to contrast the two points of view represented in these survey papers. Collberg advocates that "protection mechanisms are typically based on the application of obfuscating code transformations that add complexity" and "we hope to provide long-term security by overwhelming the adversary's analytical abilities with randomized, unique, and varying code variants". In other words, Collberg reflects the conventional wisdom in applied cryptography that successful obfuscation comes from complex

programs. On the other hand, Barak highlights "the fundamental shift from 'security through obscurity' to 'security through simplicity'." Indeed, the recent theoretical solutions to obfuscation all work by transforming programs into a more uniform format (*e.g.,* an encoding of a universal machine and/or a circuit or a matrix branching program). From the point of view of CFG: any obfuscation technique based on circuits or matrix branching programs has a trivial CFG (more details later). The aim of this paper is to explore this concept of "security from simplicity" in detail, from the point of view of practical obfuscation. We also survey several approaches to obfuscation and to show how they can be viewed through the lens of "security from simplicity".

In this paper, we focus on obfuscating control flow in programs. There have been a number of methods suggested in the literature to hide control flow, such as opaque predicates (*i.e.,* dummy branches), function splitting and recombination, and control flow flattening [9, 13, 23]. Most of these methods only protect against a static analysis. A powerful class of attacks is to execute a program in a symbolic or concolic execution tool [25, 26, 4, 22]. Hence, there is a need to develop dynamic protections to hide control flow. We propose a novel idea that may be appropriate in some settings. Our main idea is a form of "branch merging" that allows replacing code in conditional expressions with a single code segment that computes all branches in a data-dependent way. As with many other obfuscation ideas, our approach does add an overhead to the program size and running time.

We stress that we are not proposing a general-purpose obfuscation tool. In this paper, we do not consider protecting data (*e.g.,* secret keys or other constants) in programs or other types of assets. Instead, our focus is entirely on hiding the CFG of a certain type of program segment. Future work will combine this technique with other obfuscation techniques to give a more general and powerful tool.

## 2 Definitions

The literature contains many formulations of obfuscation, some of which are more precise and formal than others. Since we do not give a formal security analysis in the current version of this paper, we do not give formal definitions. We define an *asset* to be a piece of data or code that we wish to be protected (see chapter 5 of Collberg and Nagra [16]). For example, an asset might be a secret key that is used in a program (such as in white-box cryptography) or it might be a piece of code that implements some valuable proprietary algorithm. For protection of an asset to be a feasible goal, it is necessary that the asset not be easily learned by executing the program. For example, one cannot protect the secret key in a one-time pad, since it is easy to learn the key from a single message-ciphertext pair.

Giving a formal model for a non-learnable asset for a class of programs seems to be difficult, and a general formulation of such a model has not been given in the theoretical community. However, theoreticians have considered some special cases. An example is *evasive functions* [11, 6]. Informally, a function $F : X \to \{0,1\}$ is called *evasive* if it is hard to find $x \in X$ such that $f(x) = 1$ (actually, the definition is usually the other way around: a set $\mathcal{F}$ of functions is evasive if, given $x$, it is hard to find $f \in \mathcal{F}$ with $f(x) = 1$). If $P$ is a program that computes an evasive function then the asset is the condition that is being checked by $P$ (alternatively, the asset is one or more inputs $x$ that satisfy o condition: often once enough inputs are known then one can deduce the condition by some kind of "interpolation"). A famous example is the program that checks if the input $x$ is equal to some fixed value $pw$ (a password check program, also called a *point function*).

In this paper, the asset we are interested in protecting is the logical structure of the program, represented by a CFG. Recall that a CFG is a representation of a program that illustrates all if/while/switch statements and visualises the blocks of code covered by the cases as black boxes.

An obfuscator $\mathcal{O}$ takes as input a program $P$ and returns a new program $\mathcal{O}(P)$. The basic requirements of an obfuscator $\mathcal{O}$ are *(approximate) correctness*, which is that $\mathcal{O}(P)(x) = P(x)$ for all (or almost all) inputs $x$, and *polynomial slowdown*, which is that the running time of $O(P)$ should be at worst a polynomial function of the running time of $P$.

One can then define various notions of security for an obfuscator. One strong notion is *Virtual Black Box (VBB)* security (see Barak, Goldreich, Impagliazzo, Rudich, Sahai, Vadhan, Yang [7]). This is defined as: an adversary, given an obfuscated program, can learn nothing more than an adversary given only oracle access to the program. This essentially means that the obfuscated program reveals nothing more than can be known by looking at input-output pairs for the program.

There exist classes of programs $P$ and obfuscators $\mathcal{O}$ that achieve the VBB notion. One example is the set of programs $P$ that compute an *evasive* function of a certain type, such as password check programs. One can obfuscate the program $P(x) = (\text{is } x = pw)$ using a hash function as $\mathcal{O}(P)(x) = (\text{is } H(x) = c)$ where $c = H(pw)$ is hard-coded into the program. One can show that this obfuscator has VBB security in the random oracle model.

We now define the kind of obfuscator we are interested in.

**Definition 1.** Let $\mathcal{P}$ be a class of programs. A *control flow obfuscator* $\mathcal{O}$ for $\mathcal{P}$ takes as input a program $P \in \mathcal{P}$ with CFG $G$ and returns a new program $\mathcal{O}(P)$ for which the control flow graph $G'$ is very different to $G$. A *control flow de-obfuscator* adversary takes as input $\mathcal{O}(P)$ and succeeds if it can determine a CFG $G''$ that is topologically equivalent to $G$. The control flow obfuscator is *secure* if there is no efficient control flow de-obfuscator that succeeds with high probability over all programs $P \in \mathcal{P}$.

We do not claim that the CFG of a program represents total comprehension of the program design. It is quite possible that knowledge of the CFG of a program still does not allow an attacker to steal intellectual property or insert a malicious piece of code. Conversely, we also do not claim that hiding the CFG implies the obfuscation is successful. It might be the case that an obfuscator hides the control flow successfully but does not prevent the leakage of some data that is intended to remain secret. Our intention is simply the following: if there is a developer who believes that the CFG of a class of programs $\mathcal{P}$ is an asset worth protecting, then we consider obfuscation tools that allow the developer to protect that asset.

It does not make sense to try to obfuscate programs that are too simple. The correct formalism for obfuscation is to consider classes of programs. To have a meaningful definition, it is necessary that the class of programs be large, otherwise there is no IP to protect: an attacker can list all programs in the class and choose the one that gives the correct outputs on certain inputs. This is analogous to a cryptosystem needing to have a large number of possible keys. Similarly, some programs are *learnable* from input-output pairs. A simple example of a learnable function is the decryption algorithm for the one-time pad. For another example, suppose a program $P(x)$ computes a secret polynomial $a_0 + a_1 x + \cdots + a_d x^d$. Then, given $d + 1$ pairs of $(x, P(x))$, an attacker can determine the polynomial using interpolation. Finally, some program segments do not contain any asset worth protecting (*e.g.,* they are mainly concerned with I/O and can be easily generated by any programmer). Hence, the correct target for obfuscation is classes of program segments that are: large; contain an asset; are unlearnable.

## 3 Survey of Previous Work on Obfuscation Tools and Attacks

The conventional wisdom in obfuscation, surveyed well in Collberg, Thomborson and Low [17] and Collberg and Nagra [16] is to add complexity to a program and hence to make reverse engineering more difficult.

The current state of the literature is surveyed by Schrittwieser, Katzenbeisser, Kinder, Merzdovnik and Weippl [23]. Their paper recalls several standard control flow protections. One is *opaque predicates*, which are if statements inserted into the code, with complex conditional expressions that are hard to analyse but that always evaluate to either true or false. Hence, the program (either in the if clause or the else clause) may contain code segments that are never executed. A static analysis of the CFG would include such a branch and the corresponding code blocks. In other words, the obfuscator takes as input a program $P$ with CFG $G$ and returns a new program $\mathcal{O}(P)$ that has a CFG $G'$ with many more conditional components and program blocks. Since there are infinitely many possible CFGs $G'$ arising in this way, it might seem that the obfuscation process is good. However, such predicates pose no obstacle to dynamic analysis as the dummy code segments are never executed.

A powerful class of dynamic attacks is to execute a program using a symbolic or concolic execution tool [25, 26, 4, 22]. In particular, we highlight the work of Yadegari, Johannesmeyer, Whitely, Debray [26]. They study automated de-obfuscation tools whose goal is to compute a simplified CFG of a program that has been obfuscated to hide the control flow. Their work demonstrates that making a complex CFG (even when there are an extremely large number of possible CFGs of the obfuscation) is not sufficient to imply good quality obfuscation.

The ability of a symbolic tool to handle opaque predicates depends on how well it responds to the complex conditional expression. Further investigation into this issue is worthwhile, but our working assumption in this paper is that a combination of symbolic analysis and more straightforward dynamic execution can be used to strip out opaque predicates from an obfuscated program.

Another standard technique is *function splitting and recombination* [23]. Splitting is where a conditional branch is introduced into a program block with both branches leading to equivalent (but not identical) blocks of code. The complexity of the CFG is increased but the functionality unchanged. Inspired from array merging [18], using *function recombination*, two or more functions are merged into a single block so that a static analysis of the code leads to a simplified CFG [20]. However, only one set of instructions is executed and so the original CFG can be determined under a dynamic attack.

Finally, *control flow flattening* [9, 13, 23] is a general approach to hiding the CFG from a static attack. The idea is to replace the entire program or program segment with a single master switch statement and to govern control flow using certain variables whose values are updated in the various program blocks.

A good explanation is given by Cappaert and Preneel [12, 13]. Consider a program with $n$ code blocks (defined to be pieces of code that contain no branches or conditional expressions) that are linked by loops or jumps or conditional branches. An equivalent way to organise the problem is to have a central *dispatcher node (DN)*: the program starts with a call to the dispatcher node DN, which in turn calls one of the $n$ code blocks, after which control returns to the DN. The execution of the program continues in this way, until the DN decides to terminate the execution.

Control flow flattening was proposed for obfuscation by several researchers [14, 24]. Cappaert and Preneel [13] give a number of techniques to strengthen the basic idea, but they emphasise that these protections are only *static*. A detailed obfuscation tool has been developed by Blazy and Trieu [9].

Control flow flattening is a powerful tool to protect against a static analysis of the source code, but it provides no protection from symbolic tools or other dynamic attacks: when the program is executed then it is clear that certain blocks of code are executed in a certain order.

The work of Yadegari *et al.* is one of the starting points of our research: it may be the case that complex CFGs can be easily simplified, but what about the reverse process? Suppose an obfuscation tool yields a *simplified* CFG $G'$. Then, there are many CFGs $G$ that are more complicated than $G'$ and yet could have been transformed to give $G'$. The de-obfuscator now has a different task to the traditional setting: instead

of reducing a large number of complicated graphs $G'$ to a fairly well-defined minimal graph $G$, the de-obfuscator now has to expand a simple graph $G'$ into the "right one" of infinitely many possible original programs with graphs $G$. Indeed, it is not even clear if the task of the de-obfuscator is very well-defined. Hence, the goal of this paper is to develop obfuscation tools so that a simple CFG does not imply it is easy to learn the asset from the program.

# 4 Approaches to Security from Simplicity

In this section, we consider some other approaches, that try to protect programs without resulting in much more complex control flow.

## 4.1 Use of Table Lookups

White-box cryptography is a general approach to secure programs that contain a secret. A typical example is a secret-key decryption program. The program should take as input ciphertexts and return plaintexts. The cipher algorithm is known to the adversary, so the only asset in the program is the secret key. Hence, the goal is to protect the asset. A typical approach is to replace operations by lookup tables.

In this situation, since the algorithm is public, there is no interest in protecting the control flow. But it is worth pointing out that the process of replacing the decryption program with a sequence of table lookups also fits the theme of "security from simplicity": the obfuscated or white-box program just looks like a sequence of lookups in "random" tables. The security (actually, there is no security – as all these proposals are broken) is not coming from the "complexity" of the program but instead on the "uniformity" of it: two such white-box programs for two different keys "look more or less the same".

In particular, a program segment illustrated as Algorithm 1 can be re-written as simply $y := F(x)$ for some function $F$ which is more complex than $P$, $f$ and $g$. If the bit-length of $x$ is small enough then $F$ could be represented as a table. Then, the functionality of the functions $f$ and $g$, as well as the predicate $P$, are completely concealed from an automated tool.

**Algorithm 1** A simple program with an if-else condition

```
1: if P(x) then
2:     y := f(x)
3: else
4:     y := g(x)
5: end if
```

## 4.2 Matrix Branching Programs and Kilian's Randomisation

This is the theoretical tool used by Garg *et al.* [19] for their celebrated construction of iO from multilinear maps. Note that the work of Garg *et al.* is about obfuscating circuits, which already do not contain conditional statements and so the CFG is a single block. However, the ideas have been developed for Turing machines and more general computational models such as RAM programs in a number of works [1, 8, 10, 21]. It is beyond the scope of this paper to survey these papers, however we merely comment that they all reduce the problem to the case of obfuscating circuits. One approach is to write the "next step" function of the Turing machine as a circuit and to obfuscate that (this approach is taken in [8, 21]). Another approach is to use fully homomorphic encryption and run a universal Turing machine on encrypted inputs to generate

an encryption of the program output; one then writes the decryption algorithm for the FHE scheme as a circuit and obfuscates that [1]. All constructions fit within the context of security from simplicity: The obfuscated program appears to be "uniform" and the CFG of the resulting program is independent of the specific application being obfuscated.

We now briefly sketch the approach to obfuscate circuits [19], which is the starting point for all this work. The first ingredient is Barrington's theorem. This allows to take a function $f : \{0,1\}^w \to \{0,1\}$ and implement it by giving a sequence of $n$ pairs of matrices $A_{i,j}$ for $1 \leq i \leq n$ and $j \in \{0,1\}$ and a selection function $s : \{1,\ldots,n\} \to \{1,\ldots,w\}$. To compute the function $f(x_1,\ldots,x_w)$, a user computes

$$\prod_{i=1}^{n} A_{i,S(i)}.$$

If $A$ is the identity matrix then output 0 and if $A$ is some fixed non-identity matrix then output 1. Note that the length $n$ of the program may be very long even if the original circuit computing $f$ is not so big (officially, this only works for $\mathrm{NC}^1$ circuits). Note also that Barrington's theorem gives specific values for the matrices $A_{i,j}$, depending on the circuit being computed. So that one could "read off" the circuit by looking at the matrices. Hence, this alone is not sufficient for an obfuscation.

The second ingredient is Kilian's randomisation, which allows to replace the matrices $A_{i,j}$ with "random looking" matrices. The function evaluation is exactly the same process.

Garg *et al.* [19] apply the Barrington theorem to a "universal circuit $C(x,y)$, where $x$ can be thought of as specifying the "program" to be obfuscated and $y$ the "input" by a user. More generally, $\{C(x,\cdot) : x \in X\}$ could be a specific class of programs to be obfuscated.

Section 3 of [19] describes the above system as an approach to obfuscation, and discusses some attacks. The attacks are somewhat theoretical, but to get a stronger system they then use multilinear maps. Both the matrix branching program from Barrington's theorem, and the further obfuscated version using multilinear maps, have no conditional statements and so have trivial CFG, as did the original circuit being obfuscated.

Ananth, Gupta, Ishai and Sahai [2] have given a more efficient approach, that replaces Barrington's theorem and matrix branching programs with more relaxed notion and more compact construction. These approaches to obfuscation are currently very inefficient [3], so we do not discuss them further.

# 5 Proposed Approach

We now explain our approach to obfuscating control flow, which may be viewed as a form of recombination.

## 5.1 Turning Conditional Expressions into Algebraic Polynomials

We now explain how some programs that compute functions and contain conditional statements can be turned into functions without explicit conditional statements.

To give the basic idea of our method we consider the following simple example. Algorithm 2 is a program that computes income tax as a function of income, where there is a tax-free band, a middle-range tax band, and a high-income tax band.[1]

The idea of our obfuscation is to utilise bits that indicate the inequalities $(income < 35000)$ and $(income < 80000)$. This can be done by computing the values $(income - 35000)$ and $(income - 80000)$.

[1] Of course this program can easily be reverse-engineered by looking at the output on some chosen inputs, hence there is no major interest in trying to obfuscate such a program. The point of this example is only in the context of hiding control flow.

**Algorithm 2** A tax calculator

1: **if** $income \leq 35000$ **then**
2: $\quad tax = 0$
3: **else if** $income \leq 80000$ **then**
4: $\quad tax = (income - 35000) * 0.25$
5: **else**
6: $\quad tax = 11250 + (income - 80000) * 0.4$
7: **end if**

Suppose integers are represented in two's complement format, and let $SignBit$ denote the bit that is set to 0 if the integer is $\geq 0$ and 1 if the integer is $< 0$. One can then construct an algebraic polynomial that computes the same function by adding disjoint cases together. We give the obfuscated version of the program for this example in Algorithm 3.

**Algorithm 3** An obfuscated version of tax calculator

1: $s1 = SignBit(income - 35000)$
2: $s1 = SignBit(income - 80000)$
3: $tax = s1 * (1 - s2) * (income - 35000) * 0.25 + s1 * s2 * (11250 + (income - 80000) * 0.4)$

One can check that if $income \leq 35000$ then $s1 = s2 = 0$ and then $tax = 0$. If $35000 < income \leq 80000$ then $s1 = 1, s2 = 0$ and so $tax = (income - 35000) * 0.25$. Finally, if $income > 80000$ then $s1 = s2 = 1$ and the final formula is computed.

The control flow graph of Algorithm 3 is trivial, in comparison to the control flow graph of the original program. Hence, we have achieved our goal of hiding control flow.

One could also apply further transformations to this program, for example the formula could be expanded in a different way, or additional clauses that never give any contribution could be added, such as $s * (1 - s) * (income - 50000) * 0.3$.

To make reverse engineering more difficult one could add opaque predicates as expressions and exploit the distributive and commutative property of addition and multiplication to obfuscate the program further.

We remark that a similar idea is mentioned in Chapter 4 of Cappaert's thesis [12] (see "Step 2: Use a Single Uniform Statement" on page 62). However, Cappaert applies this only to the computation of the switching variable in an implementation of control flow flattening, and this is intended to make "local" static analysis more complicated. Our approach is to apply this to conditional statements themselves, removing any explicit control flow and defending against dynamic and symbolic attacks.

We now discuss some more complex examples.

First, we discuss a sorting algorithm. The basic operation in any sort algorithm is a conditional swap (see Algorithm 4).

**Algorithm 4** A conditional swap in bubble sort

1: **if** $A[i-1] > A[i]$ **then**
2: $\quad t = A[i-1]$
3: $\quad A[i-1] = A[i]$
4: $\quad A[i] = t$
5: **end if**

This can be replaced by Algorithm 5.

**Algorithm 5** An obfuscated version of the conditional swap in bubble sort

1: $s = SignBit(A[i] - A[i-1])$
2: $t1 = A[i-1]$
3: $t2 = A[i]$
4: $A[i-1] = s * t2 + (1-s) * t1$
5: $A[i] = s * t1 + (1-s) * t2$

Now, consider binary search for a value $S$ in an array $A[1], ..., A[n]$ of sorted integers (see Algorithm 6).

**Algorithm 6** Binary search

1: $L = 1$
2: $R = n$
3: **while** $L \leq R$ **do**
4: $\quad i = [(L+R)/2]$
5: $\quad$ **if** $S == A[i]$ **then return** $i$
6: $\quad$ **else if** $S < A[i]$ **then**
7: $\quad\quad R = i - 1$
8: $\quad$ **else if** $S > A[i]$ **then**
9: $\quad\quad L = i + 1$
10: $\quad$ **end if**
11: **end while**

Algorithm 6 can be written as Algorithm 7, which no longer contains any conditional statement. Notice that if $S = A[i]$ then $s1 = s2 = 0$ and so lines 7 and 8 evaluate to $L = R = i$. Subsequent iterations then compute $i = (L+R)/2 = i$ and so the loop terminates with $L = R = i$.

**Algorithm 7** Obfuscated Binary search

1: $L = 1$
2: $R = n$
3: **while** $L \leq R$ **do**
4: $\quad i = [(L+R)/2]$
5: $\quad s1 = SignBit(S - A[i])$
6: $\quad s2 = SignBit(A[i] - S)$
7: $\quad L = s1 * (1-s2) * L + s2 * (1-s1) * (i+1) + (1-s1) * (1-s2) * i$
8: $\quad R = s2 * (1-s1) * R + s1 * (1-s2) * (i-1) + (1-s1) * (1-s2) * i$
9: **end while**

Finally, the while loop can be replaced by a for loop of fixed size $\lceil \log_2(n) \rceil$, which is an upper bound on the number of iterations.

## 5.2 More Complex Programs

In future work we will describe more interesting classes of programs that can be handled using this technique. We will develop an obfuscation compiler that can be used to provide efficient obfuscations with

simplified CFG. We will also consider the performance of symbolic tools as a de-obfuscation tool against such programs.

# 6 Conclusions

In this short note we have sketched a new approach to obfuscating control flow against a dynamic or symbolic deobfuscation tool. Our ideas are aligned with the philosophy of "security from simplicity", which is a new theme in the field of obfuscation.

# References


[1] Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, and Mark Zhandry, *Differing-inputs obfuscation and applications*, IACR Cryptology ePrint Archive **2013** (2013), 689.

[2] Prabhanjan Vijendra Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai, *Optimizing obfuscation: Avoiding Barrington's theorem*, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014 (Gail-Joon Ahn, Moti Yung, and Ninghui Li, eds.), ACM, 2014, pp. 646–658.

[3] Sebastian Banescu, Martín Ochoa, Nils Kunze, and Alexander Pretschner, *Idea: Benchmarking indistinguishability obfuscation - A candidate implementation*, Engineering Secure Software and Systems - 7th International Symposium, ESSoS 2015, Milan, Italy, March 4-6, 2015. Proceedings (Frank Piessens, Juan Caballero, and Nataliia Bielova, eds.), Lecture Notes in Computer Science, vol. 8978, Springer, 2015, pp. 149–156.

[4] Sebastian Banescu, Martín Ochoa, and Alexander Pretschner, *A framework for measuring software obfuscation resilience against automated attacks*, 1st IEEE/ACM International Workshop on Software Protection, SPRO 2015, Florence, Italy, May 19, 2015 (Paolo Falcarin and Brecht Wyseur, eds.), 2015, pp. 45–51.

[5] Boaz Barak, *Hopes, fears, and software obfuscation*, Commun. ACM **59** (2016), no. 3, 88–96.

[6] Boaz Barak, Nir Bitansky, Ran Canetti, Yael Tauman Kalai, Omer Paneth, and Amit Sahai, *Obfuscation for evasive functions*, Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings (Yehuda Lindell, ed.), Lecture Notes in Computer Science, vol. 8349, Springer, 2014, pp. 26–51.

[7] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang, *On the (im)possibility of obfuscating programs*, J. ACM **59** (2012), no. 2, 6.

[8] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang, *Succinct randomized encodings and their applications*, Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015 (Rocco A. Servedio and Ronitt Rubinfeld, eds.), ACM, 2015, pp. 439–448.

[9] Sandrine Blazy and Alix Trieu, *Formal verification of control-flow graph flattening*, Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016 (Jeremy Avigad and Adam Chlipala, eds.), ACM, 2016, pp. 176–187.

[10] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan, *Succinct garbling and indistinguishability obfuscation for RAM programs*, Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015 (Rocco A. Servedio and Ronitt Rubinfeld, eds.), ACM, 2015, pp. 429–437.

[11] Ran Canetti, Guy N. Rothblum, and Mayank Varia, *Obfuscation of hyperplane membership*, Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings (Daniele Micciancio, ed.), Lecture Notes in Computer Science, vol. 5978, Springer, 2010, pp. 72–89.

[12] Jan Cappaert, *Code obfuscation techniques for software protection*, Ph.D. thesis, Katholieke Universiteit Leuven, 2012.

[13] Jan Cappaert and Bart Preneel, *A general model for hiding control flow*, Proceedings of the 10th ACM Workshop on Digital Rights Management, Chicago, Illinois, USA, October 4, 2010 (Ehab Al-Shaer, Hongxia Jin, and Marc Joye, eds.), ACM, 2010, pp. 35–42.

[14] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A Zakharov, *An approach to the obfuscation of control-flow of sequential computer programs*, International Conference on Information Security, Springer, 2001, pp. 144–155.

[15] Christian Collberg, *Engineering code obfuscation*, Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I (Marc Fischlin and Jean-Sébastien Coron, eds.), Lecture Notes in Computer Science, vol. 9665, Springer, 2016, pp. XVIII–XIX.

[16] Christian Collberg and Jasvir Nagra, *Surreptitious software: Obfuscation, watermarking, and tamperproofing for software protection*, Addison-Wesley, 2009.

[17] Christian Collberg, Clark Thomborson, and Douglas Low, *A taxonomy of obfuscating transformations*, Computer Science Technical Reports 148, 1997.

[18] Christian Collberg, Clark Thomborson, and Douglas Low, *A taxonomy of obfuscating transformations*, Tech. report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[19] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters, *Hiding secrets in software: a cryptographic approach to program obfuscation*, Commun. ACM **59** (2016), no. 5, 113–120.

[20] Matthias Jacob, Mariusz H Jakubowski, and Ramarathnam Venkatesan, *Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings*, Proceedings of the 9th workshop on Multimedia & security, ACM, 2007, pp. 129–140.

[21] Venkata Koppula, Allison Bishop Lewko, and Brent Waters, *Indistinguishability obfuscation for Turing machines with unbounded memory*, Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015 (Rocco A. Servedio and Ronitt Rubinfeld, eds.), ACM, 2015, pp. 419–428.

[22] Andreas Moser, Christopher Krügel, and Engin Kirda, *Exploring multiple execution paths for malware analysis*, 2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA, IEEE Computer Society, 2007, pp. 231–245.

[23] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl, *Protecting software through obfuscation: Can it keep pace with progress in code analysis?*, ACM Computing Surveys **49** (2016), no. 1, 4:1–4:40.

[24] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson, *Software tamper resistance: Obstructing static analysis of programs*, Tech. report, Technical Report CS-2000-12, University of Virginia, 12 2000, 2000.

[25] Babak Yadegari and Saumya Debray, *Symbolic execution of obfuscated code*, Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM New York, 2015, pp. 732–744.

[26] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray, *A generic approach to automatic deobfuscation of executable code*, IEEE Symposium on Security and Privacy, 2015, pp. 674–691.