A Tutorial on Software Obfuscation

Sebastian Banescu
Alexander Pretschner
Department of Informatics, Technische Universität München

Abstract

Protecting a digital asset once it leaves the cyber trust boundary of its creator is a challenging security problem. The creator is an entity which can range from a single person to an entire organization. The trust boundary of an entity is represented by all the (virtual or physical) machines controlled by that entity. Digital assets range from media content to code, and include items such as: music, movies, computer games and premium software features. The business model of the creator implies sending digital assets to end-users – such that they can be consumed – in exchange for some form of compensation. A security threat in this context is represented by malicious end-users, who attack the confidentiality or integrity of digital assets, in detriment to digital asset creators and/or other end-users. Software obfuscation transformations have been proposed to protect digital assets against malicious end-users, also called Man-At-The-End (MATE) attackers. Obfuscation transforms a program into a functionally equivalent program which is harder for MATE to attack. However, obfuscation can be use both for benign and malicious purposes. Malware developers rely on obfuscation techniques to circumvent detection mechanisms and to prevent malware analysts from understanding the logic implemented by the malware. This chapter presents a tutorial of the most popular existing software obfuscation transformations and mentions published attacks against each transformation. We present a snapshot of the field of software obfuscation and indicate possible directions, which require more research.

1 Introduction

A common business model for commercial media content and software creators is to distribute digital assets (e.g. music, movies, proprietary algorithms in software executables, etc.) to end-users, in exchange for some form of compensation. Even with ubiquitous cloud-based services, digital asset creators still need to ship media content and client applications to end-users. For both performance and scalability reasons, software developers often choose to develop *thick client* applications, which contain sensitive code and/or data. For example, games or media players are often thick clients offering premium features or content, which should only be accessible if the end-user pays a license fee. Sometimes, the license is temporary and therefore the client software should somehow restrict access to these features and content once the license expires. Moreover, some commercial software developers also want

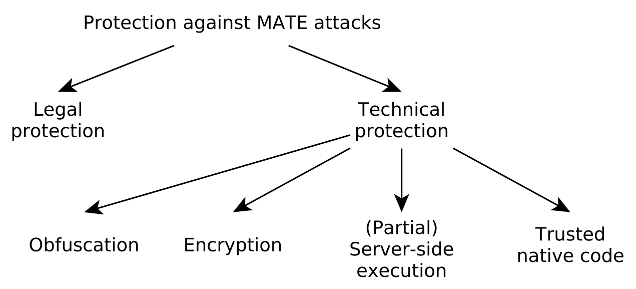

Figure 1: Classification of protections against MATE attacks proposed in [40].

to protect secret algorithms used in their client software, which give them an advantage over their competitors.

One open challenge in IT security is protecting digital assets once they leave the cyber trust boundary of their creator. The creator of digital assets can range from a single person to an organization. The security threat in this context is represented by malicious end-users, who among other things, may want to:

- Use digital assets without paying the license fees required by the creators of that digital asset.
- Redistribute illegal copies of digital assets to other end-users, sometimes in order to make a profit.
- Make changes to the digital assets (e.g. by tampering with its code), in order to modify its behavior.

Such malicious end-users are also called Man-At-The-End (MATE) attackers [37], and they have control of the (physical or virtual) machine where the digital asset is consumed. Practically, any device under the control of an end-user (e.g. PC, TV, game console, mobile device, smart meter, etc.), is exposed to MATE attacks. A model of the MATE attacker capabilities, akin to the degree of formalization of the Man-In-The-Middle (MITM) attacker introduced by Dolev-Yao [51], is still missing from the literature. However, MATE attackers are assumed to be extremely powerful. They can examine software both statically using manual or automatic static analysis, or dynamically using state of the art software decompilers and debuggers [89]. Shamir *et al.* [113] present a MATE attack, which can retrieve a secret key used by a black-box cryptographic primitive to protect the system if it is stored somewhere in non-/volatile memory. Moreover, the memory state can be inspected or modified during program execution and CPU or external library calls can be intercepted (forwarded or dropped) [133]. Software behavior modifications can also be performed by the MATE attacker by tampering with instructions (code) and data values directly on the program binary or after they are loaded in memory. The MATE attacker can even simulate the hardware platform on which software is running and alter or observe all information during software operation [35]. The only remaining line of defense in case of MATE attacks is to increase the complexity of an implementation to such an extent that it becomes economically unattractive to perform an attack [35].

Researchers, practitioners and law makers have sought several solutions for this challenge, all of which have their advantages and disadvantages. Figure 1 shows a classification of these solutions, proposed by Collberg et al. [40]. On the one hand, there are legal protection frameworks that apply to some geographic regions, such as the Digital Millennium Copyright Act [41] in the USA, the EU Directive 2009/24/EC [84], etc. On the other hand, there are technical protection techniques (complementing legal protection), which are divided into four subcategories, namely: (1) software based obfuscation, (2) encryption (via trusted hardware), (3) server-side execution and (4) trusted (i.e. tamper-proof or tamper-evident) native code. The latter three subcategories will be briefly discussed in the related work section. The *obfuscation* subcategory is the main focus, i.e. software-only protection that does not rely on trusted entities.

An obfuscator is in essence a compiler that takes a program as input, and outputs a functionally equivalent program, which is harder to understand and analyze than the input program. The meaning of the phrases “functionally equivalent” and “harder to understand and analyze” will be discussed in this chapter. For instance, some classical compiler optimizations are also considered obfuscation transformations [40], because in order to make the code more efficient, such optimizations may replace control-flow abstractions that are easy to understand by developers (e.g. loops), with other constructs which are less straightforward (e.g. goto statements).

This chapter presents a tutorial of several popular obfuscation transformations together with illustrative examples. It also mentions the MATE attacks published in the literature, which have been proposed for defeating each obfuscation transformation. The rest of the chapter is structured as follows. Section 2 presents classification dimensions for obfuscation transformations. Section 3 presents classification dimensions for MATE attacks. Section 4 presents a survey of obfuscation transformations and state of the art MATE attacks that claim to break each obfuscation. Section 5 discusses the current state of software obfuscation versus MATE attacks. Section 6 presents related work, and section 7 concludes the chapter.

2 Classification of Code Obfuscation Transformations

Several surveys and taxonomies for software obfuscation have been proposed in literature [8, 40, 86, 94, 109]. This section describes the classification dimensions presented in those works and discusses their advantages, disadvantages and overlaps. We present the classification dimensions in increasing order of importance, starting with the least important category.

2.1 Abstraction Level of Transformations

One common dimension of code transformations is the level of abstraction at which these transformations have a noticeable effect, i.e. *source code*, *intermediate representation* and *binary machine code*. Such a distinction is relevant for usability purposes, e.g. a JavaScript developer will mostly be interested in source code level transformations and a C developer will mainly be interested in binary level. However, none of the previously mentioned taxonomies and surveys classify transformations according to the abstraction level. This is due to the fact that some obfuscation transformations have an effect at multiple abstraction levels.

Moreover, it is common for papers to focus only on a specific abstraction level, disregarding transformations at other levels.

2.2 Unit of Transformations

Larsen et al. [86] proposed classifying transformations according to the of granularity at which they are applied. Therefore they propose the following levels of granularity:

- *Instruction level* transformations are applied to individual instructions or sequences of instructions. This is due to the fact that at the source code level, a code statement can consist of one or more IR or Assembly instructions.
- *Basic block level* transformations affect the position of one or more basic blocks. Basic blocks are a list of sequential instructions that have a single entry point and end in a branch instruction.
- *Loop level* transformations alter the familiar loop constructs added by developers.
- *Function level* transformations affect several instructions and basic blocks of a particular subroutine. Moreover, they may also affect the stack and heap memory corresponding to the function.
- *Program level* transformations affect several functions inside an application. However, they also affect the data segments of the program and the memory allocated by that program.
- *System level* transformations target the operating system or the runtime environment and they affect how other programs interact with them.

The unit of transformation is important in practice because developers can choose the appropriate level of granularity according to the asset they must protect. For example, loop level transformations are not appropriate for hiding data, but they are appropriate for hiding algorithms. However, the same problem, as for the previous classification dimension, arises for the unit of transformation, namely the same obfuscation transformation may be applicable to different units of transformation.

2.3 Dynamics of Transformations

The dynamics of transformation – used by Schrittwieser et al. [109] – indicate whether a transformation is applied to the program or its data *statically* or *dynamically*. Static transformations are applied once during: implementation, compilation, linking, installation or update, i.e. the program and its data does not change during execution. Dynamic transformations are applied at the same times as static transformations, however, the program or its data also change during loading or execution, e.g. the program could be decoded at load time, because it was encoded on disk. Even though dynamic code transformations are generally considered stronger against MATE attacks than static ones, they require the code pages to be both writable and executable, because the code may modify itself during execution. This opens the door for remote attacks (e.g. code injection attacks [122]), which are more dangerous for end-users than MATE attacks. Moreover, dynamic transformations generally have a higher performance overhead than static transformations, because code has to first be written (generated or modified) and then executed. Therefore, on the one hand, many benign software developers avoid dynamic transformations entirely. On the other hand, dynamic transformations are heavily used by malware developers, because they are not generally concerned about high performance overhead.

2.4 Target of Transformations

The most common dimension for classifying obfuscation transformations is according to the target of transformations. This dimension was first proposed by Collberg et al. [40], who indicated four main categories: layout, data, control and preventive transformations. In a later publication Collberg and Nagra [39] refined these categories into four broad classes: abstraction, data, control and dynamic transformations. Since the last class of Collberg and Nagra [39] (i.e. dynamic transformations), overlaps with the dynamics of transformation dimension, described in subsection 2.3, we will use a simplification of these two proposals where we remove the dynamic transformations class and merge the abstraction, layout and control classes. Therefore, the remaining transformation targets are:

- *Data transformations*, which change the representation and location of constant values (e.g. numbers, strings, keys, etc.) hard-coded in an application, as well as variable memory values used by the application.
- *Code transformations*, which transform the high-level abstractions (e.g. data structures, variable names, indentation, etc.) as well as the algorithm and control-flow of the application.

This dimension is important for practitioners, because it indicates the goal of the defender, i.e. whether the defender wants to protect data or code. Note that obfuscation transformations which target data may also affect the layout of the code and its control-flow, however, their *target* is hiding data, not code. In practice data transformations are often used in combination with code transformations, to improve the resilience of the program against MATE attacks.

Data transformations Data transformations can be divided into two subcategories:

1. *Constant data* transformations, which affect static (hard-coded) values. Abstractly, such transformations are encoding functions which take one or more constant data items i (e.g. byte arrays, integer variables, etc.), and convert them into one or more data items $i' = f(i)$. This means that any value assigned to, compared to and based on i is also changed according to the new encoding. There will be a trade-off between resilience on one hand, and cost on the other, because all operations performed on i require computing $f^{-1}(i)$, unless f is homomorphic w.r.t. those operations.
2. *Variable data* transformations, which modify the representation or structure of variable memory values. The goal of such transformations is to hamper the development of

Dimension	Possible values
Abstraction level	Source code
	Intermediate representation
	Binary machine code
Unit	Instruction
	Basic block
	Loop
	Function
	Program
	System
Dynamics	Static
	Dynamic
Target	Constant Data
	Variable Data
	Code Logic
	Code Abstraction

Table 1: Classification dimensions for obfuscation transformations.

automated MATE attacks, which can assume that a certain variable memory value will always have a certain representation or a certain structure (e.g. an integer array of 100 contiguous elements). If such assumptions hold then automated attacks are easier to develop, because they mainly rely on pattern matching. Therefore, by employing the transformations presented in this section, one can raise the bar for these kinds of attacks.

Note that variable transformations are not the transformations that affect the names or the order of variables, but the representation of the data stored in those variables.

Code transformations Code transformations can also be divided into two subcategories, namely:

1. *Code logic* transformations, which affect the control-flow of the program, its ease of readability and maintainability. Such transformations are aimed at adding complexity to the code in order to prevent MATE attackers from understanding what the algorithm(s) implemented by the code are.

2. *Code abstraction* transformations, which destroy programming abstractions, are generally the first hints that MATE attackers use to start reverse engineering a program.

As opposed to *constant data transformations*, which in some cases perform one-way mappings of data to a completely different domain which requires no interpretation, code transformations must always perform a mapping to the same domain of executable code, because obfuscated code must be able to run on the underlying machine where it is installed.

2.5 Summary of Obfuscation Transformation Classification

Table 1 provides a summary of the classification dimensions described above along with the possible discrete values that each dimension can take. In the next section we choose to present a survey of obfuscation transformations classified according to their *target of transformation*, because it entails a clear partition of transformations.

3 Classification of MATE Attacks

In contrast to the classification of software protection techniques (see section 2), the classification of MATE attacks has been the topic of relatively few publications [2, 37, 109]. This section presents the most relevant classification dimensions of MATE attacks.

3.1 Attack Type Dimension

Basile et al. [19] argue that it is not feasible to consider every possible attacker goal since it represents the desired end-result for the attacker, e.g. see the position of other players in computer games, or play premium content without paying, etc. Therefore, in this chapter we classify attacks according to their type, i.e. the means through which an attacker goal can be achieved. According to Collberg et al. [36, 37], there are four types of information a MATE attacker may be interested in recovering from an obfuscated program:

- The original or a simplified version of the *source code*. This is always the case for MATE attackers who are interested in intellectual property theft, i.e. stealing a competitor's algorithm.

- A statically embedded or dynamically generated *data item*. Common examples of such data items are decryption keys used by Digital Rights Management (DRM) technologies to play premium content only on authorized devices, for authorized users. However, data items may also include hard-coded passwords, IP addresses, etc.

- The sequence of obfuscation transformations and/or tools used to obfuscate (protect) the program, also called *metadata*. This kind of information is often used by antivirus engines to detect suspicious binaries, based on the fact that several previously seen malware have used the same obfuscation transformations and/or tools.

- The *location*, (i.e. lines of code or bytes) of a particular function of the code. For instance, the attacker may be interested in the module which performs premium content decryption in order to copy it and reuse it in another program, without necessarily understanding how it works.

We compare these four information types proposed by Collberg et al. [36, 37], with the *analyst's aims* proposed by Schrittwieser et al. [109]:

- *Code understanding*, which according to its description in the paper maps to the *source code* information type described above. However, the name of this category suggests a more general attack type than a full recovery of the entire *source code*, because it could be sufficient to have a partial code understanding. For example, a malware analysis engine can decide that a software is malicious using its observed behavior (e.g. unsolicited calls to premium telephone numbers), which does not require full understanding of the source code. Moreover, *metadata* recovery also falls inside of this category of *code understanding*. Therefore, in this chapter we will use this more general category, i.e. *code understanding*.

- *Finding the location of data*, which maps perfectly onto the *data item* information type described above. However, the phrase *location of data* may be mistaken for the *location* information type. Therefore, this chapter will simply use *data item recovery.*

- *Finding the location of program functionality*, which maps onto the *location* information type described above. However, Schrittwieser et al. [109] also add that this type of information may be used to answer questions regarding if the program is malicious or not. In this chapter we move such questions to the *code understanding* category, because we believe an answer to such a question, requires some level of code understanding, but not necessarily recovering the entire *source code.*

- *Extraction of code fragments*, which does not directly map onto any of the information items described above. However, we believe that *finding the location of program functionality* is a prerequisite to this aim of the analyst, because extraction can only be done after the location of the code fragment has been recovered. Therefore, in this chapter we associate this aim of the analyst with the *location* information type.

In sum, we use the following three categories of attack types with the meanings discussed above: (1) *code understanding*, (2) *data item recovery* and (3) *location recovery.*

3.2 Dynamics Dimension

Dynamics is one of the most commonly used classification criteria for automated MATE attacks and it refers to whether the attacked program is executed on a machine or not, i.e.:

- *Static analysis* attacks do not execute the program on the underlying (physical or virtual) machine. The subject of the analysis is the static code of the program.

- *Dynamic analysis* attacks run the program and record executed instructions, function calls and/or memory states during execution, which are the subject of analysis.

Static attacks are commonly faster than dynamic attacks. However, static analysis attacks do not handle many code obfuscation transformations as well as dynamic analysis attacks. This does not mean that dynamic analysis attacks can handle any kind of obfuscation easily. For instance, it is challenging to dynamically analyze programs employing code transformation techniques that achieve *temporal diversity*, i.e. the program has significantly different execution traces and/or memory states on every execution. Moreover, dynamic analysis is in general incomplete, i.e. it cannot explore or reason about all possible executions of a program, as opposed to static analysis.

3.3 Interpretation Dimension

Code interpretation refers to whether the program's code or the artifacts generated using it (e.g. static disassembly, dynamic traces), are treated as text or are interpreted according to a semantic meaning (e.g. operational semantics). Therefore the two types of code interpretation considered in this chapter are:

- *Syntactic attacks* which treat the program's code or any other artifacts generated by executing or processing it, as a string of bytes (e.g. characters). For example, pattern matching on static code [114] and pattern recognition via machine learning traces of instructions [15], treat the code as a sequence of bytes.

- *Semantic attacks* which interpret the code according to some semantics, e.g. denotational semantics, operational semantics, axiomatic semantics and variations thereof [59]. For example, abstract interpretation [44] uses denotational semantics, while fuzzing [120] uses operational semantics.

Syntactic attacks are generally faster than semantic attacks due to the missing layer of abstraction that interprets the code. Coincidentally, most syntactic attacks are performed via static analysis and most semantic attacks are performed via dynamic analysis. However, there are exceptions e.g. the syntactic analysis of dynamically generated execution traces and semantic static analysis via abstract interpretation [44].

3.4 Alteration Dimension

Alteration refers to whether the automated MATE attack changes (alters) the code or not. This type of classification is analogous to the *message alteration* classification of MITM attacks on communication channels. Hence there are two types of code alteration:

- *Passive attacks* do not make any changes to the code or data of the program. For instance, extracting a secret key or password from a program does not require any code alterations.

- *Active attacks* make changes to the code or data of a program. For example, removing data or code integrity checks (e.g. password checks), requires modifying the code of the program. Also "disarming" malware may also involve tampering with its code.

If an attacker's goal can be achieved via either passive or active attacks, then the type of attack used depends on the complexity of the program under attack and the types of protection the program has in place. For instance, if a program is protected via dynamically verified checksums of the program input, then active attacks require finding and disabling these checksumming instructions, which could be more costly than a passive attack.

3.5 Summary of MATE Attack Classification

Table 2 provides a summary of the classification dimensions described above along with the possible discrete values that each dimension can take. In the remainder of this chapter we will refer to these dimensions when describing attack implementations.

4 Survey of Obfuscation Transformations

Obfuscation transformations can be implemented in different ways, i.e. the obfuscation transformation gives only a high-level description (e.g. pseudo-code) and it leaves it up to the

Dimension	Possible values
Attack type	Code understanding
	Data recovery
	Location recovery
Dynamics	Static
	Dynamic
Code interpretation	Syntactic
	Semantic
Alteration	Passive
	Active

Table 2: Classification dimensions for automated MATE attacks.

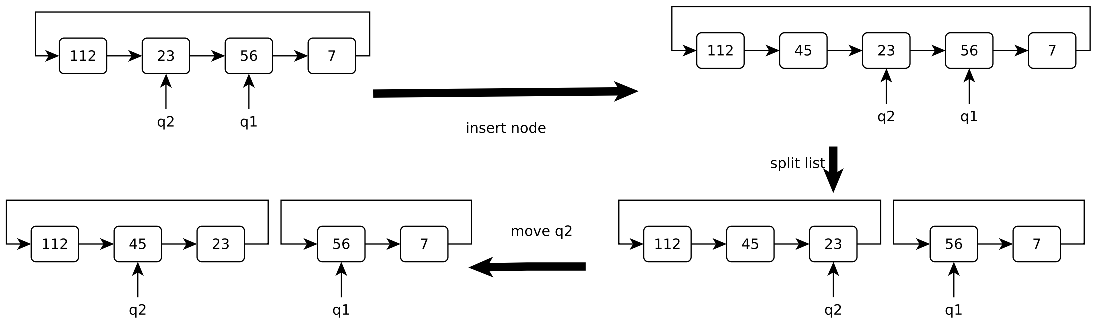

Figure 2: Opaque expressions based on linked lists.

obfuscation engine developer to take concrete implementation decisions. This section provides a description of the abstract idea behind common obfuscation transformations, it does not focus on any particular implementation of an obfuscation engine. This state of the art survey groups obfuscation transformation techniques according to their target of transformation, namely data and code. We also give examples using code snippets written in C, JavaScript and Assembly language, to illustrate the transformations.

4.1 Constant Data Transformations

Opaque predicates Collberg et al. [40] introduce the notion of *opaque predicates.* The truth value of these opaque predicates is invariant w.r.t. the value of the variables which comprise it, i.e. opaque predicates have a value which is fixed by the obfuscator e.g. the predicate $x^2 + x \equiv 0 \pmod 2$ is always true. However, this property is hard for the attacker to deduce statically. Collberg et al. [40] also present an application of opaque predicates, which is called *extending loop condition.* This is done by adding an opaque predicate to loop conditions, which does not change the value of the loop condition, but makes it harder for an attacker to understand when the loop terminates.

Opaque predicates can be created based on mathematical formulas which are hard to solve statically, but they can also be built using any other problem which is difficult to compute statically, e.g. aliasing. Aliasing is represented by a state of a program where a certain memory location is referenced by multiple symbols (e.g. variables) in the program. Several works in literature show that pointer alias analysis (i.e. deciding at any given point during

Listing 1: Code before Encode Literals

```c
int main(int ac, char* av[]) {
  int a = 1;
  // do stuff
  return 0;
}
```

Listing 2: Code after Encode Literals

```c
int main(int ac, char* av[]) {
  double s = sin(atof(av[1]));
  double c = cos(atof(av[1]));
  int a = (int) (s * s + c * c);
  // do stuff
  return 0;
}
```

execution, which symbols may alias a certain memory location), is undecidable [71, 85, 104]. Therefore, Collberg et al. [40] propose to leverage this undecidability result to build opaque predicates using pointers in linked lists. For instance, consider the linked list illustrated in the top-left part of Figure 2. This circular list consists of four elements and it has two pointers (i.e. q_1 and q_2) referencing its elements. After performing three list operations, i.e. inserting another list element (top-right part of Figure 2), splitting the list in two parts (bottom-right) and then moving the pointer q_2 two elements forward (bottom-left), the obfuscator knows that the element referenced by q_1 is larger than the element referenced by q_2. However, this relation is hard to determine using static analysis techniques, therefore $q_1 > q_2$ represents an opaque predicate, which is always true. Wang et al. [128] employ such opaque expressions to hide code pointer values, hence, obfuscating control flow via data obfuscation.

One extension of opaque predicates was made by Palsberg et al. [99], who propose *dynamic opaque predicates* which change their truth values between different runs of the program. A further extension appeared in the work of Majumdar and Thomborson [90], who proposed *distributed opaque predicates* which change their truth values during the same execution of a program, depending on the location in code, where they are evaluated. This means that the values of the opaque predicate changes during execution of a program due to values being sent and received from other programs in a distributed system.

Attacks: Due to their popularity, opaque predicates have been the target of several MATE attacks published in the literature. Dalla Preda et al. [47] propose a location and data recovery attack which based on abstract interpretation [44], hence it is a static, semantic and passive attack. Banescu et al. [9, 12] propose a data recovery and code understanding attack based on symbolic execution, which is dynamic, semantic and passive. This attack also aimed other obfuscation transformations which will be presented later in this section such as: converting static data to procedural data, encoding arithmetic, inserting dead code, virtualization and control flow flattening. Salem and Banescu [107] propose a code understanding attack in order to identify which programs have been obfuscated using opaque predicates and which have been obfuscated using the same obfuscation transformations enumerated for the previous attacks, plus *program encoding*, which is presented later in this section.

Convert static data to procedural data (a.k.a. Encode Literals) A simple way of obfuscating a hard-coded constant is to convert it into a function (program) that produces the constant at runtime [40]. This transformation implies choosing an invertible function (program) f, feeding the constant to f as input and storing the output. During runtime the inverse of that function, i.e. f^{-1} is applied to the output of f which was stored somewhere in the program. Obfuscating a hard-coded constant value (e.g. 5), by using simple encoding functions (e.g. $f(i) = a \cdot i + b$), leads to small execution overheads. However, since i is

Listing 3: Hiding the value of $k = 0x876554321$ using Mixed Boolean-Arithmetic.

```
int main(int argc, char* argv[]) { // compiled on a 32-bit architecture
 int x = atoi(argv[1]);
 int x1 = atoi(argv[2]);
 int x2 = atoi(argv[3]);

 int a = x*(x1 | 3749240069);
 int b = x*((-2*x1 - 1) | 3203512843);
 int d = ((235810187*x+281909696- x2) ^ (2424056794+x2));
 int e = ((3823346922*x+3731147903+2*x2) | (3741821003 + 4294967294*x2));

 int k = 135832444*d +4159134852*e+272908530*a+409362795*x+136454265*b+2284837645 +
    415760384*a*b+ 2816475136*a*d+1478492160*a*e+3325165568*b*b+2771124224*b*x + 1247281152*
    a*x+1408237568*b*d+2886729728*b*e+4156686336*x*x+4224712704*x*d + 415760384*a*a
    +70254592*x*e+1428160512*d*d+1438646272*d*e+1428160512*e*e;
 // do stuff
 return 0;
}
```

a constant, such functions can also be deobfuscated using compiler optimizations such as constant folding [7]. Therefore, another way of hiding constants is to build expressions dependent on external variables (e.g. user input). For instance, *opaque expressions* – similar to opaque predicates except that their value is non-Boolean – always have a certain fixed value during program execution, e.g. $cos^2(x) + sin^2(x)$ is always equal to 1, regardless of the value of x. Therefore, the constant value 1 from the C code from Listing 1, can be encoded using this opaque expression, which cannot be simplified away by the compiler. The resulting code after this obfuscation is shown in Listing 2. This transformation can also be applied to string constants, which can be split into substrings or even single characters, which can be interpreted as integers. At runtime these substrings or characters would be concatenated in the right order to form the original string.

Attacks: This obfuscation transformation has been successfully attacked by the two semantic attacks of Banescu et al. [9,12] based on symbolic execution and the attack of Salem and Banescu [107] based on pattern recognition, mentioned in the attacks on opaque predicates.

Mixed Boolean-Arithmetic Zhou et al. [140], propose a data encoding technique called Mixed Boolean-Arithmetic (MBA). MBA encodes data using linear identities involving Boolean and arithmetic operations, together with invertible polynomial functions. The resulting encoding is made dependent on external inputs such that it cannot be deobfuscated using compiler optimization techniques. The following example is taken from [140] and it aims to encode an integer value $k = 0x87654321$. The example gives k as an input to the following second degree polynomial with coefficients in $\mathbb{Z}/(2^{32})$:

$$f(x) = 727318528x^2 + 3506639707x + 6132886 \pmod{2^{32}}.$$

The output of computing $f(k)$ is 1704256593. This value can be inverted back to the value of k during runtime by using the following polynomial:

$$f^{-1}(x) = 1428291584x^2 + 1257694419x + 4129091678 \pmod{2^{32}}.$$

Zhou et al. [140] describe how to pick such polynomials and how to compute their inverse. Since the polynomial $f^{-1}(x)$ does not depend on program inputs and the value of $f(k)$ is hard-coded in the program, an attacker can retrieve the value of k by using constant propagation. In order to create a dependency of $f^{-1}(k)$ on program inputs, the following Boolean-arithmetic identity is used:

$$2y = -2(x \vee (-y-1)) - ((-2x-1) \vee (-2y-1)) - 3.$$

This identity makes the computation of a constant value (i.e. $2y$, which is the value of $f(k)$ in the running example), dependent on a program input value, i.e. x. Note that this relation can be applied multiple times for different program inputs. The resulting Boolean-arithmetic relation is further obfuscated by applying the following identity:

$$x + y = (x \oplus y) - ((-2x-1) \vee (-2y-1)) - 1.$$

Making the computation of $f^{-1}(k)$ dependent on three 32-bit integer input arguments of the program and applying the second Boolean-arithmetic relation multiple times gives the code in Listing 3, which dynamically computes the original value of $k = 0x87654321$. Note that in Listing 3, variables a, b, d and e are input dependent, common subexpressions of the MBA expression of k.

Attacks: Guinet et al. [67] propose a static analysis tool called Arybo, which is able to simplify MBA expressions. Their attack is meant for code understanding and data recovery and it uses code semantics to achieve this goal.

White-box cryptography This transformation was pioneered by Chow et al. [32, 33], who proposed the first White-Box Data Encryption Standard (WB-DES) and White-Box Advanced Encryption Standard (WB-AES) ciphers in 2002. The goal of White-Box Cryptography (WBC) is the secure storage of secret keys (used by cryptographic ciphers), in software, without hardware keys or trusted entities. Instead of storing the secret key of a cryptographic cipher separately from the actual cipher logic, white-box cryptography embeds the key inside the cipher logic. For instance, for Advanced Encryption Standard (AES) ciphers, the key can be embedded by multiplication with the T-boxes of each encryption round [58]. However, simply embedding the key in the T-boxes of AES is prone to key extraction attacks since the specification of AES is publicly known. Therefore, WB-AES implementations use complex techniques to prevent key extraction attacks, e.g., wide linear encodings [135], perturbations to the cipher equations [26] and dual-ciphers [79].

The idea behind the white-box approach in [32] is to encode the internal AES cipher logic (functions) inside Look-up Tables (LUTs). One extreme and impractical instance of this idea is to encode all plaintext-ciphertext pairs corresponding to an AES cipher with a 128-bit key, as a LUT with 2^{128} entries, where each entry consists of 128-bits. Such a LUT would leak no information about the secret-key but exceed the storage capacity of currently available devices. However, this LUT-based approach also works for transforming internal AES functions (e.g. XOR functions, AddRoundKey, SubBytes and MixColumns [58]) to table lookups, which can be divided such that they have a smaller input and output size. For instance, Listing 4 shows the implementation of a 8-bit XOR gate, which takes one byte value as an input argument and outputs the bitwise-XOR of this value and the Most Significant

Listing 4: 8-bit XOR with MSB of secret key

```
char xor(char input) {
  return input ^ 0x53;
}
```

Listing 5: LUT-based 8-bit XOR

```
 1 char lut[256] = {
 2   0x53, 0x52, 0x51, ..., 0x5C,
 3   0x43, 0x42, 0x41, ..., 0x4C,
 4   0x73, 0x72, 0x71, ..., 0x7C,
 5     |                      |
 6   0xA3, 0xA2, 0xA1, ..., 0xAC
 7 };
 8
 9 char xor(char input) {
10   return lut[input];
11 }
```

Byte (MSB) of the secret key of the AES cipher instance, which is 0x53 in Listing 4. This function can easily be converted to a LUT-based implementation – as illustrated in Listing 5 – by constructing a LUT containing all input-output combinations for the 8-bit XOR function from Listing 4. Therefore, this LUT contains 256 elements and the LUT-based version of the XOR function simply requires a look-up in this table, as shown on line 10 of Listing 5.

LUTs can also be used to encode random invertible bijective functions, which are used to further obfuscate the LUTs representing internal AES functions. This is necessary because an attacker could extract the Most Significant Byte (MSB) of the secret key from the LUT in Listing 5. In order to hide the key, WBC proposes to generate a random permutation of 256 bytes and apply it to the LUT. To be able to use the resulting LUT, the inverse permutation would be composed with the next operation following the XOR during AES encryption. Since the attacker would not know the randomly generated permutation value, s/he would no longer be able to extract the key directly from the LUT. Converting several steps of an AES cipher to LUT-based implementations and then applying random permutations to these LUTs leads to an implementation which is much more compact than the huge LUT with 2^{128} entries of 128-bits. Typically, the size of a WB-AES cipher is around a few megabytes. The same idea can also be applied to other ciphers as well.

Attacks: Several MATE attacks on WB-DES [133, 134] and WB-AES [23, 48] have been published in the literature, on the ground of algebraic attacks, which treat each cipher as an overdefined system of equations [43]. All of these attacks assume that the structure and purpose of the LUTs is known. Therefore, Banescu et al. [14] propose using data obfuscation techniques to hide the location and structure of the LUTs used by WBC ciphers.

One-way transformations One-way transformations refer to mapping data values from one domain to another domain (e.g. $f(i) = i'$), without needing to perform the inverse mapping $f^{-1}(i')$ during runtime. This means that f must be homomorphic w.r.t. the operations performed using i. For instance, a cryptographic hash function such a Secure Hash Algorithm (SHA), e.g. SHA-256 (denoted H) may be used as a one-way transformation. H can map a hard-coded string password s to a 256-bit value, i.e. $v = H(s)$. Generally, the only operation performed with a hard-coded password is an equality check with an external user input i. Hence, v does not need to be mapped back to s during runtime. Instead, the program can compute $v' = H(i)$ and verify the equality between the hard-coded value v and the dynamically computed v'. Since the implementation of H for cryptographic hash functions, does not disclose the inverse mapping H^{-1}, the MATE attacker is forced to either guess s,

Listing 6: Split variable

```
1 char b1, b2, b3, b4; // i = b1,b2,b3,b4
2
3 int add(int a) {
4   return a + ((b1 << 8 + b2)
5                  << 8 + b3)
6                  << 8 + b4;
7 }
```

Listing 7: Merged variables

```
int i; // i = b1,b2,b3,b4
char add_b1(char a) {
  return a + (i >> 24);
}
//...
char add_b4(char a) {
  return a + (i & 0xff);
}
```

or identify the equality comparison and modify it such that it always indicates equality regardless of i. The latter tampering attack can be hampered if the code of the equality check is highly obfuscated, which can be achieved by applying code obfuscation transformations (see subsection 4.3).

Attacks: More than a decade ago, Wang et al. [129] suggested how to find collisions in hash functions such as MD4, MD5, HAVAL 128 and RIPEMD. However, it was only recently that Stevens et al. [117] discovered the first collision for the SHA1 hash function. Both of these attacks are meant to recover input data for the hash function, which hashes to the same value as another input data. They are dynamic and semantic since they require executing the hash functions.

4.2 Variable Data Transformations

Split variables The idea behind this transformation is to substitute one variable by two or more variables [40]. For instance, a 32-bit integer variable i can be split into four byte variables b_1, b_2, b_3 and b_4 that represent the integer i, i.e. $i = b_1 \cdot 256^3 + b_2 \cdot 256^2 + b_3 \cdot 256 + b_4$. This idea is similar to converting static data to procedural data, except that splitting variables does not apply to constant values, but to any value that a variable may hold at any moment during execution. Listing 6 shows a C-code snippet illustrating the previously mentioned example of replacing an integer by four bytes. This code snippet also shows that such an obfuscation transformation requires us to implement functions even for simple arithmetic operations such as addition (lines 3-7). We must reconstruct the 32-bit integer value before performing the actual arithmetic operation, i.e. addition. This kind of function must be implemented for all operations, which are needed in the original un-obfuscated program.

Attacks: Slowinska et al. [115] propose a data recovery attack based on dynamic trace analysis of so-called *temporal reuse intervals*, which indicate the usage patterns of certain memory locations. This attack is also applied to the merging variables transformation presented next.

Merge variables Two or more variables can be merged into a single variable, if the ranges of the combined variables fit within the precision of the compound variable [40]. For example, up to four 8-bit variables can be packed into a 32-bit variable, i.e. the inverse operation than *splitting variables*. Listing 7 shows an illustration of this example, where four byte variables, namely b_1, b_2, b_3 and b_4 have been merged into an integer variable i. Any operations on the individual variables has to be carefully crafted in order not to affect the other variables.

Listing 8: Folded array

```
int folded[10][10] = {{1, 2, ..., 10},
                      {11, 12, ..., 20},
                      ...
                      {91, 92, ..., 100}};
```

Listing 9: Flattened array

```
int flattened[100] = {1, 2, ..., 100};
```

For instance, arithmetic addition functions – as shown on lines 2-8 of Listing 7 – must be implemented for each of the four different variables. These functions could be further obfuscated using the code obfuscation transformations in order to raise the bar for attackers. A MATE attacker would first need to understand these functions in order to figure out that multiple variables are stored in a compound variable.

Attacks: The previously mentioned MATE attack proposed by Slowinska et al. [115] for splitting variables, also applies to merging variables. In addition to this attack, Viticchi et al. [126] performed a user study where a group of students managed to successfully reverse engineer programs obfuscated using the merge variables transformation by employing a combination of both static and dynamic analysis tools and techniques.

Restructure arrays Similarly to variables, arrays can be split or merged [40]. However, in addition to that, arrays can be folded (increasing the number of dimensions), or flattened (decreasing the number of dimensions). For instance, if the original code contained an array of 100 integer elements, then this array could be folded into a 10 by 10 matrix as shown in Listing 8. We can also consider the dual, where the original code contained the 10 by 10 matrix and it would be flattened into a 1-dimensional array of 100 elements as shown in Listing 9. Folding and flattening break code abstractions put in by software developers (e.g. matrices are flattened into arrays), which force reverse engineers to first understand logic in the code before they can recover this useful abstraction.

Attacks: Slowinska et al. [116] propose a code understanding attack based on pattern recognition of access patterns on execution traces generated by symbolic execution. This attack mixes both static and dynamic techniques in order to recover data structures from obfuscated code. It is a passive attack which uses code semantics. This attack is not only aimed to break restructuring of arrays, but also loop transformations, presented later in this chapter.

Reorder variables This transformation changes the location or the name of variables in the program code, by permuting or substituting them [35]. Moreover, it can re-purpose variables such that they are no longer used for the same (single) task. It has low cost and it improves resilience, because automated MATE attacks can no longer assume certain

Listing 10: Original basic block

```
mov eax, 0x10h
mul ebx, eax
cmp ebx, 0x10h
je 0x12345678
```

Listing 11: Basic block after reordering variables

```
mov edx, 0x10h
mul ebx, edx
cmp ebx, 0x10h
je 0x12345678
```

patterns, e.g. that variables are always laid-out in a certain order, or that one specific variable (name) is used for a certain purpose only. An example of such a transformation is illustrated in the Assembly-code snippets from Listing 10 and Listing 11, where the register `eax` in the former listing is substituted by register `edx` in the latter listing. Pappas et al. [100] apply this transformation at the binary level by reassigning register operands at the basic block level. They show that this transformation is able to eliminate (on average) over 40% of Return Oriented Programming (ROP) gadgets in different instances of the same program. This means that using this transformation breaks 40% of the patterns in the binary code.

Attacks: Griffin et al. [65] present an automated MATE attack, which is able to perform data recovery for the purpose of malware string signature extraction and is able to break this transformation of re-ordering variables. Their attack is static and semantic.

Dataflow Flattening Dataflow flattening, proposed by Anckaert et al. [4], is an advanced version of variable reordering, inspired by the idea of oblivious RAM proposed by Goldreich and Ostrovsky [63]. It periodically reorders data stored on the heap via a Memory Management Unit (MMU), such that the functionality of the program is not altered. It is not feasible to show a meaningful code snippet for the MMU that would fit in one page of this chapter, but the intuition behind the way in which it reorders data on the heap is simple.

1. The MMU allocates a new memory region on the heap, for a given variable.
2. It copies the value of that variable to the new memory region.
3. The MMU updates all pointers from the old to the new memory region of the variable.
4. Finally, it deallocates the old memory region.

In addition to reordering the data on the heap, dataflow flattening also proposes moving all local variables from the stack to the heap and scrambling pointers to hide the relation between the different pointers returned to the program. This transformation has a resilience against MATE attacks, nonetheless, its execution overhead is also high.

Attacks: At the time of writing this chapter, we are not aware of reported attacks on this obfuscation technique in the literature.

Listing 12: Original function prologue and epilogue

```
push ebp     ;save previous stack frame
mov ebp, esp ;base of this stack frame
...          ;function body
mov esp, ebp ;discard this stack frame
pop ebp      ;restore previous frame
ret
```

Listing 13: Prologue and epilogue after stack randomization

```
sub esp, 0x43
push ebp
sub esp, 0x5f
mov ebp, esp
...
mov esp, ebp
add esp, 0x5f
pop ebp
add esp, 0x43
ret
```

Listing 14: Original addition function

```
int a = 5;
int b = 10;
...
int sum = a + b;
```

Listing 15: Addition function obfuscated with DSR

```
int mask_a = rand();
int a = 5 ^ mask_a;
int mask_b = rand();
int b = 10 ^ mask_b;
...
int mask_sum = rand();
int sum = ((a ^ mask_a) + (b ^ mask_b)) ^
     mask_sum;
```

Randomized stack frames This transformation assigns each newly allocated stack frame a random position on the stack [61]. For this purpose it subtracts a random value from the stack pointer in the function prologue (simulating a push of multiple elements) and adds this value before the function returns in the function epilogue. Additionally, Fedler et al. [55] propose padding each stack frame internally by a random amount, such that return address and local variables are at random offsets. Note that, these random offsets could be generated at compile time or during runtime. This is again done by performing random subtractions from the stack pointer in the function prologue and undoing these subtractions in the function epilogue. A typical function prologue and epilogue is shown in the Assembly code snippet from Listing 12. The prologue starts by saving the address of the base of the previous stack frame (line 1). Then it sets the base of the new stack frame (line 2). After the body of the function is executed the contents of the new (current) stack frame are discarded (line 4) and the previous stack frame is restored (line 5). Listing 13 shows the prologue and epilogue after the stack randomization transformation is applied. Note the instructions inserted on lines 1, 3, 7 and 9, which subtract and add random values – `0x43` and `0x5f` in our example – from the stack pointer register (i.e. `esp`). Since the subtracting and adding instructions are stack operation, the value subtracted in the prologue, must be added in reverse order in the epilogue. These added instructions can be easily identified by the MATE attacker, however, they could be further obfuscated using both code and data transformations.

Attacks: Strackx et al. [118] propose a data recovery attack that is able to bypass the protection offered by randomized stack frames via a technique called a *buffer overread* (note that this is different from buffer overflow). This attack is dynamic, semantic and active because it changes the data of the program in process memory.

Data space randomization Cadar et al. [27] and Bhatkar and Sekar [21] introduce a data transformation technique which they call Data Randomization and Data Space Randomization (DSR), respectively. The idea of these two techniques is to XOR (i.e. encrypt) data values stored in program memory (e.g. stack, heap, etc.) with randomly generated masks. The masks do not need to be fixed, they can be generated dynamically at runtime and used to encrypt the data values. Whenever a data value must be read by the program, it is first decrypted using the right mask. After an authorized modification of a decrypted data value occurs, the result is re-encrypted with the same or with a different mask depending on the implementation. The technique is inspired by PointGuard [45], which encrypts code pointers. DSR offers protection against MATE attackers who want to extract or modify data values from/in process memory. One challenge of implementing DSR is that different pointers may point to the same encrypted memory value, therefore, they must use the same mask to decrypt a data value. This problem is solved in part by performing a static alias analysis of the code [5], before the transformation is applied. We have already mentioned that alias analysis is in general undecidable, therefore, an approximation is performed and a common mask is used for all pointers that cannot be statically determined.

Listing 14 shows a C-code snippet performing a simple summation operation. Listing 15 shows how the function from Listing 14, after it is obfuscated using DSR. Note that each value is XOR-ed with a random mask, which must be used for decryption when performing the addition operation on line 7 of Listing 15. Even though the actual values of the variables are now hidden by masking, the security issue is shifted to hiding the masks or the relation between values and masks from MATE attackers. This can be achieved by applying other obfuscation transformations on top of DSR. DSR is reported to introduce an average runtime overhead of 15% and it can protect against buffer- and heap-overflow attacks.

Attacks: The attack of Strackx et al. [118] which was presented as an attack for the *random stack frames* transformation is also able to bypass DSR.

4.3 Code Logic Transformations

Instruction reordering This technique targets sequences of instructions, which when permuted, do not alter the original program execution [35]. Similarly to variable reordering, this transformation is meant to break MATE attacks based on pattern matching. However, it has very low resilience w.r.t. human-assisted MATE attacks, because it does not increase the difficulty of code understanding by much. An example of instruction reordering is shown in Listing 17, where the instructions on lines 1 and 2 have been reordered from their original positions in Listing 16. The candidate instruction sequences targeted by this technique are also candidates of parallel processing optimizations, because they can be independently performed by different execution threads, without any danger of race conditions. The reordered sequence of instructions must be equivalent to the original sequence. The cost of this transformation are low. Pappas et al. [100] have employed instruction reordering on binary basic block level and have shown that this transformation reduces the number of ROP gadgets by over 30%, hence, increasing the resilience against ROP attacks. Note that this transformation can be also performed at basic block level, however, this would have a lower increase in resilience compared to instruction reordering.

Attacks: Zhang et al. [139] propose a static code understanding attack in order to detect repackaged Android applications, which are suspected to be malicious. Their attack uses code semantics to build a so-called *view graph* of each application, which is compared to other applications in order to determine if they are repackaged versions of the same application. The authors indicate that this attack is resilient to multiple code obfuscation transformations

Listing 16: Code before instruction reordering

```
1 mov eax, ebx
2 add ecx, edx
3 add eax, ecx
4 ...
```

Listing 17: Code after instruction reordering

```
1 add ecx, edx
2 mov eax, ebx
3 add eax, ecx
4 ...
```

Listing 18: Assembly code#1 performing swap

```
mov edx, eax
mov eax, ebx
mov ebx, edx
```

Listing 19: Assembly code#2 performing swap

```
push eax
push ebx
pop eax
pop ebx
```

presented in this section, namely: merging functions, opaque predicates, inserting dead code, removing functions, function argument randomization and converting static data to procedural.

Instruction substitution This technique (first mentioned in [35]) is based on the fact that in some programming languages as well as in different Instruction Set Architectures (ISAs), there exist several (sequences of) equivalent instructions. This means that substituting an instruction (sequence) with its equivalent will not change the semantic behavior of the program, nevertheless, it will result in a different binary representation. A concrete implementation and evaluation of this technique is presented by Jacob et al. [74] and it is also used in the *Hydan* tool [52]. Listing 18 shows an Assembly code snippet representing a swap function from register `eax` to register `ebx`, using register `edx` as an auxiliary variable. Listing 19 shows one of the many possible instruction sequences – equivalent to Listing 18 – presented by Jacob et al. [74]. The transformation has a moderate cost, however, it offers low resilience against MATE attacks, due to the fact that the number of transformations available is limited. Regarding the resilience against remote attacks, Pappas et al. [100] measured the effect of this transformation at binary basic block level, against ROP attacks and discovered that it reduces less than 20% of ROP gadgets. Moreover, the use of uncommon instructions will decrease stealth, i.e. indicate to an attacker where the substitution occurred. In order to improve the stealth of this transform, De Sutter et al. [49] proposed a technique called *instruction set limitation*, which proposes candidates for substitution based on the statistical distribution of instruction types in the program. Mason et al. [92] also proposed a similar technique with the purpose of improving the stealth of shellcode by encoding it as text written in the English language.

Attacks: The code understanding and data recovery attack of Banescu et al. [9], presented as an attack for opaque predicates is also applicable to bypass this transformation.

Listing 20: Code before Encode Arithmetic

```
int main(int ac, char* av[]) {
  int x = atoi(av[1]);
  int y = atoi(av[2]);
  int w = atoi(av[3]);
  int z = x + y + w;
  // do stuff
  return 0;
}
```

Listing 21: Code after Encode Arithmetic

```
int main(int ac, char* av[]) {
  int x = atoi(av[1]);
  int y = atoi(av[2]);
  int w = atoi(av[3]);
  int z = (((x ^ y) + ((x & y) << 1)) | w) +
          (((x ^ y) + ((x & y) << 1)) & w);
  // do stuff
  return 0;
}
```

Listing 22: Code before inserting garbage code

```
int sum = 0;
for (i = 0; i < arr_len; i++)
  sum += arr[i];
int average = sum / arr_len;
```

Listing 23: Code after inserting garbage code

```
1 int sum = 0;
2 int prod = 1;
3 for (i = 0; i < arr_len; i++) {
4   sum += arr[i];
5   prod *= arr[i];
6 }
7 int average = sqrt(prod);
8 average = sum / arr_len;
```

Encode Arithmetic This technique is proposed by Collberg [36] and it is a variant of *instruction substitution*, which substitutes boolean or arithmetic expressions by expressions involving both boolean and arithmetic operations, which are harder to understand. One example of such a transformation is illustrated by Listing 21, which shows a C code snippet after *encode arithmetic* has been applied to the right-hand side of the assignment to variable `z` from line 5 of Listing 20.

Attacks: The drawback of this approach is that there are a limited number of such Boolean-arithmetic identities available in the literature [130]. Eyrolles et al. [54] have proposed writing a reverse transformation for each of them, after identifying Mixed Boolean-Arithmetic (MBA) expressions via pattern matching.

Garbage insertion This technique implies inserting arbitrary sequences of instructions, that are independent of the data flow of the original program and do not affect its input-output (IO) behavior (functionality) [35]. The possible sequences that may be inserted are virtually infinite, nonetheless, the performance-cost grows proportionally to the number of inserted instructions. Listing 22 and Listing 23 shows a program that computes the average of all elements in an array `arr`, before and after garbage code has been inserted. Note that in Listing 23, lines 2, 5 and 7 represent garbage code that has no influence on the output value of the program. However, inserting garbage code changes the relative offset the original instructions of the program. It also raises the complexity of reverse-engineering by cluttering the original code. However, note that garbage code should be inserted only after performing compiler optimizations, because it can be identified and eliminated via taint analysis.

Attacks: Performing the generic attack based on taint analysis, proposed by Yadegari et al. [136], would remove lines 2, 5 and 7 in Listing 23, because the final value of `average` (line 8 in Listing 23) has no data dependency on `prod`. The transformation space for this technique is limited only by physical or practical run-time constraints such as time delays and memory consumption, because as opposed to dead code, garbage code is always executed.

Insert dead code This transformation was first proposed by Collberg et al. [40] and it modifies the control-flow of a program such that a dead branch is added, i.e. a branch that is never taken during runtime. Adding the dead branch is facilitated by opaque predicates, because one of the branches of a conditional statement that uses such an opaque predicate, will never be executed, while the other branch will always be executed. In order not to disclose the truth value of opaque predicates by leaving the dead branch empty, Collberg suggests inserting dummy code on the dead branch. To further confuse the attacker, the dead code can be a buggy version of the other branch, which is always chosen. For instance, consider the code snippet from Listing 24 which converts the first input argument to an integer and then sets the value of variable y to the square root of the input argument's value. Listing 25 shows that dead code can be inserted anywhere by first inserting a conditional statement with an opaque predicate that is always true (line 4) and then adding dead code in the branch that is never taken (line 7). Note that we can wrap as many lines of code as we want using the conditional statement. Moreover, the size of the dead code can be arbitrarily large.

Attacks: Along with the previously presented attacks by Salem and Banescu [107], Zhang et al. [139], Yadegari et al. [136], and Banescu et al. [9], there are numerous works in the filed of compilers presenting optimizations for dead code removal [68, 81].

Adding and removing function calls These two techniques were first proposed by Cohen [35], and can be applied at any unit of transformation. Adding a call to a sub-routine implies: (1) selecting an arbitrary sequence of instructions, (2) creating a sub-routine using that sequence and (3) finally, substituting the original sequence with a call to that sub-routine. Removing a call to a sub-routine implies: (1) substitute all calls to a sub-routine with the body of that sub-routine and (2) delete the sub-routine. Listing 26 shows a code snippet that contains a function, while Listing 27 shows a code snippet where this function has been removed. The reverse transformation from Listing 27 to Listing 26, is *adding function calls*. These transformations cause changes in the structure of a program, which creates more complexity for MATE attacks. The cost of this technique grows or decreases with the number of inserted and removed sub-routine calls, respectively. This method has been extended by Banescu et al. [15], such that system calls are added or existing system calls are substituted with equivalent ones. They call this transformation *behavior obfuscation* because it hides the system call trace analyzed by behavioral malware analysis engines.

Attacks: We have already mentioned the attack of Zhang et al. [139], which also claims to bypass this obfuscation transformation. In addition to this work, many works on behavioral malware detection claim they are resilient to the addition or removal of system calls [15, 132]. Such machine learning based malware detection approaches are classified as dynamic, passive and syntactic, because they treat the function call traces generated by the obfuscated software as a sequence of bytes.

Loop transformations Several loop transformations have been proposed as compiler optimization passes by Bacon et al. [7]. Collberg et al. [40] argue that these loop transformations

Listing 24: Code before inserting dead code

```
int main(int ac, char* av[]) {
  int x = atoi(av[1]);
  int y = sqrt(x);
  // do stuff
  return 0;
}
```

Listing 25: Code after inserting dead code

```
 1 int main(int ac, char* av[]) {
 2   int x = atoi(av[1]);
 3   int y;
 4   if (x*x + x % 2 == 0)
 5     y = sqrt(x);
 6   else
 7     y = x * x;
 8   // do stuff
 9   return 0;
10 }
```

Listing 26: Add function calls

```c
int foo(int a, int b) {
  return a + b;
}
...
int c = foo(a, b + 1);
```

Listing 27: Remove function calls

```c
int c = a + b + 1;
```

Listing 28: Code with jumps removed

```
mov edx, eax
mov eax, ebx
mov ebx, edx
...
```

Listing 29: Code with jumps added

```
    mov edx, eax
    jmp L1
    ...
L2:mov ebx, edx
    ...
L1:mov eax, ebx
    jmp L2
    ...
```

also increase software complexity metrics and can therefore be considered obfuscation transformations that increase resilience against attacks. *Loop tilling* or *blocking* is intended to improve cache locality, by dividing loop iteration lengths into parts that fit in the CPU cache. This increases the nesting level of loops and is therefore more potent. *Loop distribution* or *fission* breaks the independent instructions in a loop body into multiple loops with the same iteration length, which increases the number of loops in the code. *Loop unrolling* replicates the body of the loop a certain number of times and reduces the number of iterations correspondingly, which increases the number of lines of code in the program.

Attacks: The dynamic code understanding attack by Slowinska et al. [116], already presented as an attack on the *restructure arrays* transformation, is also applicable to loop transformations.

Adding and removing jumps These techniques change the control-flow of the program by adding spurious jumps or removing existing jumps [35]. Adding jumps can be done by substituting an arbitrary sequence of instructions I by: (1) a jump to a random position, (2) followed by I and (3) a jump to the instruction immediately following I in the original version of the program. An example is illustrated in Listing 29, where the code from Listing 28 has been transformed by adding two unconditional jumps to labels `L1` and `L2`. Removing jump instructions may also be done if it does not alter the original semantics of the program, e.g. unconditional jumps may be removed and the code from the address of the jump, merged with the code preceding the unconditional jump. An example is shown in Listing 28, where all jumps from the code in Listing 29 have been removed. However, in practice adding jumps is more frequently employed in order to increase the complexity of the MATE attack. The transformation space of this technique is bounded by the length of the program it is applied to. The cost of this method grows (decreases) with the number of inserted (removed) jump instructions. The resilience of adding jumps can be increased by further obfuscating the addresses of the jumps using data obfuscation techniques such as *opaque expressions* or *converting static data to procedural data*.

Attacks: If the target of the added jumps are not made dependent on input values using opaque expressions, then they are trivial to bypass using the already mentioned dynamic attack of Yadegari et al. [136]. On the other hand removing jumps can be bypassed by dynamic taint analysis on augmented Control Flow Graphs (CFGs), proposed by Yadegari et al. [137], which is a dynamic, passive and semantic attack.

Program encoding This technique keeps one or more instructions encoded (i.e. encrypted [28, 127] or compressed [98]), while the program is not executing and decodes the sequence(s) when the program is running [35]. The resilience of program encoding against attacks depends on the algorithm used for encoding, e.g. a compression algorithm can be undone without a secret key, while an encryption requires finding the key. However, the costs may also be relatively high compared to other obfuscation techniques, because the code has to be decoded before it can be executed. There is a trade-off between resilience and cost depending on the level of granularity at which this transformation is applied, i.e. if applied at instruction level, the cost and resilience are high, because each instruction is decoded, executed and re-encoded, hence the whole code is never stored in decoded form in memory, at one point in time. While if program encoding is applied at program level, the whole code is decoded and afterwards it starts executing, hence an attacker could perform a memory dump after decoding to have a copy the whole code. Additionally, this technique does not protect well against dynamic analysis attacks, e.g. during execution the code is decoded in memory and it can be read or modified directly in memory by the MATE attacker.

Attacks: Tang et al. [121] propose a static, passive and syntactic attack against polymorphic malware, in order to extract signatures based on the position and distribution of byte values in obfuscated malware binaries. Qiu et al. [103] propose a dynamic, passive and semantic attack based on taint analysis, in order to determine the location of integrity checks inside code.

Self-modifying code This technique has been discussed in several works [35, 78, 88, 94]. It implies adding, modifying and/or removing instructions of a program during its execution. Therefore, it creates a high complexity for static-analysis attacks. Kanzaki et al. [78] proposed replacing real instructions with bogus instructions that would get replaced by real instructions before they are executed and then replaced again by bogus instructions after execution. This transformation requires a sound analysis of all possible execution paths leading to and following the instruction(s) that are to be modified. Madou et al. [88] propose to apply self-modifying code at the function level by creating so-called function templates, i.e. arrays of byte having a larger size than any single function in a chosen subset of functions. For instance, if the subset consists of functions f_1 and f_2, the code of function f_1 is

f_1

memory offset:	0	1	2	3	4
memory value:	b7	48	a0	53	fa

f_2

memory offset:	0	1	2	3
memory value:	e9	48	a0	33

T

memory offset:	0	1	2	3	4
memory value:	?	48	a0	?	?

Edit Scripts

$e_1 = [0 \to b7, 3 \to 53, 4 \to fa]$

$e_2 = [0 \to e9, 3 \to 33]$

Figure 3: Self-modifying code via function templates and edit scripts

5 bytes long and the code of function f_2 is 4 bytes long, then the template T must be at least 5 bytes long as shown in Figure 3. Note that function templates are generated by an intersection of the code bytes of all other functions in the subset, i.e. common code bytes are kept in place (e.g. 48 and a0 in Figure 3) and other code bytes are randomly initialized. Each function f is associated to an edit script e, which indicates the locations (i.e. indices) of the function template that must be patched and the values they must be patched with, in order to reconstruct the code of f. Therefore, when any function f is called, the edit script e corresponding to f is first executed and then the resulting code in the function template is executed.

Attacks: Self-modifying code can be effective against dynamic-analysis attack, which aim to break the integrity of the code (e.g. via dynamic code patching), if the executed instructions are different in different runs of the program with the same inputs. However, Nguyen et al. [96] have shown that applying such temporal diversity to instruction level encryption (i.e. instruction set randomization [18]) has negative effects on confidentiality of the encryption key because encrypting the same code with different keys leaks information about the code, similar to two-time pads [93]. In the case of self-modifying code, there is no dedicated decoder routine, as was the case for the *program encoding* transformation. Instead, different parts of the code are “responsible” for modifying other instructions and these are often spread throughout the entire program. The trade-off between resilience and cost is similar to that of *program encoding* and *virtualization*. Dalla Preda et al. [46] applied an abstract interpretation based attack to generate signatures for metamorphic malware samples, hence this attack is a code understanding, static, passive and semantic attack.

Virtualization obfuscation This technique is related to the *program encoding* technique, because it also implies an encoding of instructions [35]. Additionally, it also requires an interpretation engine (called “simulator” or “emulator”), which is able to decode the instructions and execute them on the underlying platform. The simulator may also be running on top of another simulator and so forth, giving an arbitrary nesting level. Normally, this creates complexity for static-analysis attacks, because the attacker has to first understand the custom interpreter logic and then the code running on top of it. The most significant difference of virtualization w.r.t. program encoding is that no code must be written to a memory location during decoding. However, the trade-off between resilience and cost for this transformation

Listing 30: Point function program

```
int main(int argc, char* argv[]) {  // Virtualization bytecode:
  char branch_cond = 1;             // a5 00 07

  branch_cond &= argv[1][0] == '1'; // 87 00 00 02
  branch_cond &= argv[1][1] == '2'; // 87 00 01 03
  branch_cond &= argv[1][2] == '3'; // 87 00 02 04
  branch_cond &= argv[1][3] == '4'; // 87 00 03 05
  branch_cond &= argv[1][4] == '5'; // 87 00 04 06

  if (branch_cond)                  // 1f 00 02
    printf("You win!\n");           // 03 00
  return 0;                         // 42 01
}
```

are the same as in the case of program encoding. For clarity, we provide all of the steps of the *Virtualize* transformation for the program in Listing 30. This program prints the message "You win!" on standard output if the first argument passed to this program is equal to "12345". The virtualization transformation is applied to this program using the following steps and the result is illustrated in Listing 31:

1. Map variables, function parameters and constants to entries in a common `data` array, which represents the memory of the interpreter. This array is initialized on lines 3-4 in Listing 31. Its first position represents the `branch_cond` variable from Listing 30 and the following entries represent constants such as the return value, the ASCII codes of the characters from '1' to '5' and logical `true` encoded as 1.

2. Map all statements in a function to a new randomly chosen language, which represents the *instruction set architecture* (ISA) of the interpreter. In our example the ISA is defined by:

   - Variable assignment is encoded using 3 bytes, namely the opcode (`0xa5`) and the index of the left- and right-hand operands inside the `data` array.
   - Equality comparison, followed by applying the logical *AND* operation to between the result and another variable. Examples of such instructions are shown on lines 4-5 in Listing 30. Such an instruction is encoded using 4 bytes, namely the opcode (`0x87`), the variable to which the boolean value is assigned and the two other byte values which are compared for equality.
   - Conditional branch statements are encoded using 3 bytes, namely the opcode (`0x1f`), the boolean variable which is tested and the number of bytes to jump over if the variable is false.
   - Printing a string on standard output is encoded using 2 bytes, namely the opcode (`0x03`) and the index of the string to be printed in the list of hard-coded strings of the function. In our example the list of hard-coded strings contains only one string and is defined on line 2 of Listing 31.
   - The return instruction is encoded using 2 bytes, namely the opcode (`0x42`) and the value that should be returned by the program.

   Now we can write virtualization bytecode corresponding to the C program in Figure 30, which is shown in the comments of the code from the same figure.

3. Store the encoded bytecode inside the `code` array, which is initialized on lines 5-9 in Listing 31.

4. Create an interpreter for the previously generated ISA, which can execute the instructions in the `code` array using the `data` array as its memory. The input-output behavior of this execution must be the same as that of the original program. The interpreter can be seen on lines 11-32 of Listing 31. It consists of an infinite `while` loop, which has a `switch` statement inside. Each `case` section of the `switch` statement is an opcode handler, i.e. each possible opcode in the bytecode program is processed by a dedicated

Listing 31: Point function program obfuscated with virtualization

```
 1 int main(int argc, char* argv[]) {
 2   char const *strings = "You win!\0";
 3   unsigned char data[8] = {0,    // branch_cond var
 4      0, 49, 50, 51, 52, 53, 1}; // constants
 5   unsigned char code[30] = {0xa5, 0x00, 0x07, 0x87, 0x00,
 6      0x00, 0x02, 0x87, 0x00, 0x01, 0x03, 0x87,
 7      0x00, 0x02, 0x04, 0x87, 0x00, 0x03, 0x05,
 8      0x87, 0x00, 0x04, 0x06, 0x1f, 0x00, 0x02,
 9      0x03, 0x00, 0x42, 0x01};
10   int vpc = 0;
11   while (1)
12   switch (code[vpc]) {
13     case 0xa5 : // variable assignment
14       data[code[vpc+1]] = data[code[vpc+2]];
15       vpc += 3;
16       break;
17     case 0x87 : // equality comparison plus and
18       data[code[vpc+1]] &=
19         (argv[1][code[vpc+2]] == data[code[vpc+3]]);
20       vpc += 4;
21       break;
22     case 0x1f : // if statement
23       vpc += (data[code[vpc+1]]) ? 0 : data[code[vpc+2]];
24       vpc += 3;
25       break;
26     case 0x03 : // printf string
27       printf("%s\n", strings + code[vpc+1]);
28       vpc += 2;
29       break;
30     case 0x42: // return
31       return data[code[vpc+1]];
32   }
33 }
```

part of the interpreter. The current instruction to be processed by the interpreter is indicated by an integer variable of the interpreter called the *virtual program counter* (VPC). The VPC is used to index the instructions in the `code` array and it is initialized with the offset of the first instruction in that array (line 10). In every instruction handler the operands of the current instruction are used to perform the operation(s) corresponding to this instruction. Afterwards, the VPC is set to the offset of the following bytecode instruction to be executed. This interpreter should be augmented with cases representing bogus opcodes for all possible byte values in order to increase the resilience against MATE attacks.

Together with virtualization a wide range of additional data and code obfuscation transformations can be applied. Banescu et al. [11] propose randomizing the layout of the bytecode instructions in the `code` array, as well as the format of these instructions. Collberg [36] proposes encoding the value of the VPC using opaque expressions, changing the interpreter's dispatch method from a `switch`-statement to table lookups, and many more.

Attacks: Kinder [80] proposes a static, semantic attack based on abstract interpretation and a technique called *VPC lifting*, in order to automatically recover memory values at any code location. Rolles [105] proposes a manual, static, semantic and passive attack to understand the mapping between the bytecode and the instruction handlers of the interpreter. Banescu et al. [11] present an experiment where a dynamic, active, syntactic attack, which

Listing 32: Code before Control Flow Flattening (CFF)

```
int gcd(int a, int b){
  while (a != b)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```

Listing 33: Code after CFF

```
int gcd(int a, int b){
int next = 0;
while (1) {
  switch (next) {
    case 0:
      if (a != b) next = 1;
      else next = 2;
      break;
    case 1:
      if (a > b) next = 3;
      else next = 4;
      break;
    case 2: return a;
      break;
    case 3: a = a - b; next = 0;
      break;
    case 4: b = b - a; next = 0;
      break;
    default:
      break;
}}}
```

is able to remove a large portion of the code of the interpreter and recover the original logic of the program.

Control flow flattening Wang et al. [128] and Chow et al. [34] proposed CFF, which collapses all the basic blocks of a function into a flat CFG, which hides the original control flow of the program. A program transformed by CFF is similar to an interpreter (as we saw for *virtualization obfuscation*), which chooses the right basic block where to go on the second CFG level. An example is shown in Listing 33, where the function computing the Greatest Common Divisor (GCD) of two integers a and b – shown in Listing 32 – has been transformed by CFF. As opposed to virtualization obfuscation – which uses a virtual program counter (VPC) – the order in which the cases of the `switch` statement must be executed is indicated by a control variable (i.e. `next` in the example from Listing 33), which is updated by every case of the switch statement accordingly. Updating `next` can occur before, during or after part of the logic of the original program is executed. The infinite loop is exited when a case contains a return or break statement. Another difference w.r.t. *virtualization obfuscation* is that multiple cases of the switch statement of a program transformed by CFF, may contain the same instructions. For virtualized programs each case represents a different instruction, which may appear multiple times in the original program. This means that in order to recover the original control flow of a program obfuscated using CFF, one must simply order the cases of the switch statement in the correct order, while this is not applicable to virtualized programs, because some cases may be needed multiple times.

Attacks: Udupa el al. [123] were the first to propose an automated MATE attack to recover the original control flow from a program obfuscated using CFF. Their method combined dynamic and static analysis techniques and was able to accurately recover the original control flow.

Listing 34: Code before branch functions

```
   mov edx, eax
   jmp L1
   ...
L2:mov ebx, edx
   jmp L3
   ...
L1:mov eax, ebx
   jmp L2
L3:...
```

Listing 35: Code after branch functions

```
   mov edx, eax
   push 1
   call f
   ...
L2:mov ebx, edx
   push 3
   call f
   ...
L1:mov eax, ebx
   push 2
   call f
L3:...
```

Branch functions Linn and Debray [87] propose hiding the control flow of calls, conditional and unconditional jumps, from static disassembly algorithms, by replacing them with calls to a so called *branch function*. An example is shown in Listing 35 where all jumps from the original code shown in Listing 34 have been replaced by a call to offset f, which represents the start address of the branch function. The branch function computes the actual target of the jump dynamically using a parameter passed by the callee – via `push` instructions in Listing 35 – and a lookup-table. Instead of returning to the instruction immediately following the call instruction, the branch function either jumps to the address of the original jump instruction which it replaced, or to several "junk" bytes after the call instruction that it replaced. The structure of the control flow graph is also flat similar to CFF, however, the *switch*-statement is replaced by the branch function.

Schrittwieser and Katzenbeisser [108] present an extension of the idea from [87], which is explicitly aimed at defending against both static- and dynamic-analysis techniques. The targets of the branch functions are ROP gadgets (i.e. short instruction sequences ending in a return instruction) [112]. Additionally, they generate gadget graphs which add redundancy such that the one path in the original code can have multiple paths in the obfuscated code. This is meant to hamper dynamic analysis attacks by generating different traces for the same inputs. The disadvantage of this method is that its resilience increases inversely proportional to the size of the gadgets, while its cost decreases exponentially with this size.

Attacks: Kruegel et al. [83] present a static, passive, semantic approach to bypass branch functions in order to disassemble code obfuscated using this transformation to a large extent.

4.4 Code Abstraction Transformations

Merging and splitting functions These two techniques are the code correspondents to the data obfuscation transformations of merging and splitting variables [35]. Merging is done by creating larger functions with more inputs and outputs, some of which are independent. An example is shown in Listing 37, where `func3` is the result of merging `func1` and `func2` from Listing 36. Note that `func3` uses a flag variable `c`. The body of `func1` is executed when `c` is even and the body of `func2` is executed with `c` is odd. Splitting is done by dividing large functions into smaller functions, e.g. the function in Listing 37 can be split into two functions as shown in Listing 36. Similarly to the *adding and removing calls* transformations, this technique changes the structure of the program, breaking abstractions added by developers, which makes the code more difficult to understand. Moreover, the cost of this transformation is increased or decreased with the number of split, respectively merged functions.

Attacks: Rugaber et al. [106] present ideas for implementing a static, semantic and active attack for code obfuscated by merging functions. The goal of their attack is to both detect the location of the lines of code belonging to different functions and to extract this code into separate functions.

Remove comments and change formatting This transformation is only applicable to programs which are delivered as source code (e.g. JavaScript). Comments are removed if they exist and all space, tab and newline characters are also removed, which results in a continuous string of code which is more potent against human attackers, than the original code. An example is shown in Listing 39, which is the result of removing comments and all formatting of the code from Listing 38. The original formatting and comments cannot be recovered [40], however, the code can be easily reformatted using automated tools. The cost of this transformation is low and in many cases it even improves memory costs and execution speed. Therefore, such transformations have found their way into commercial products, such as: Stunnix [119], DashO [101], Dotfuscator [102], Thicket [110], ProGuard [66] and yGuard [138]. However, the resilience against MATE attacks of these obfuscation transformations is very low because a similar alignment can be automatically generated even by free and open source Integrated Development Environments (IDEs).

Attacks: Bichsel et al. [22] propose a code understanding attack against this transformation, based on probabilistic learning of large code bases. This attack is implemented as a service called DeGuard and it is applied statically, syntactically and passively to android applications.

Scrambling identifier names This transformation implies changing all symbol names (e.g. variables, constants, functions, classes, etc.) into random strings [40]. One example is shown in Listing 41 where all the variable names of the code from Listing 40, have been changed to random identifiers. This is a one-way transformation, because the names of the symbols cannot be automatically recovered by a deobfuscator. Therefore, the MATE is forced to understand what a symbol is from a semantics point of view. It has a much higher resilience than formatting removal since identifiers contain useful abstractions added by software developers. Similarly to removing comments and changing formatting, this

Listing 36: Splitting functions

```
func1(int a, int b) {
  x = 4;
  if (a < 3)
    x = x + 6;
  x *= b;
}

func2(int a, int c) {
  y = a + 12;
  y = y/c;
}
```

Listing 37: Merging functions

```
func3(int a, int b, int c) {
  if (c % 2 == 0) {
    x = 4;
    if (a < 3)
      x = x + 6;
    x *= b;
  } else {
    y = a + 12;
    y = y/b;
  }
}
```

Listing 38: Original JavaScript code

```
function NewObject(prefix) {
  var count=0;
  // This function generates a pop-up
  this.SayHello=function(msg) {
    count++;
    alert(prefix+msg);
  }
}
var obj=new NewObject("Message : ");
obj.SayHello("You are welcome.");
```

Listing 39: JavaScript code after removing comments and formatting

```
function NewObject(prefix){var count=0;
    this["SayHello"]= function(msg){count
    ++;alert(prefix+ msg)}}var obj= new
    NewObject("Message : ");obj.SayHello(
    "You are welcome.")
```

Listing 40: Code before scrambling identifiers

```
sum = 0;
for (i = 0; i < arr_len; i++)
  sum += arr[i];
average /= arr_len;
```

Listing 41: Code after scrambling identifiers

```
za82b547bcb = 0;
for (z1c0ab7cf0c = 0; z1c0ab7cf0c < za862d19cbc;
    z1c0ab7cf0c++)
  za82b547bcb += zc1c28ca67f[z1c0ab7cf0c];
z8c8f7c7867 /= za862d19cbc
```

transformation has a very low cost and it is also used as an optimization that reduces code size, because long symbol names can be replaced by shorter ones.

Attacks: Ceccato et al. [29] investigate the relative strength of two different obfuscation transformations, namely *scrambling identifier names* and *opaque predicates*. They find that *scrambling identifier names* poses more challenges for human-assisted attacks than *opaque predicates*. This holds in the context where identifiers in the original program have a proper semantic meaning (in English). On the other hand the machine learning based attack of Bichsel et al. [22], which was already mentioned as an attack for removing comments and changing formatting, is able to automatically recover meaningful identifier names with high accuracy.

Removing library calls and programming idioms Most programs perform calls to external libraries providing useful data structures (e.g. lists, maps, etc.) and algorithms (e.g. sorting, searching, etc.). MATE attackers often start by inspecting calls made to external libraries to give a high-level indication of what the program is doing. This transformation implies replacing such dependencies on external libraries with own implementations where possible [40]. Note that such a transformation is stronger than static linking, which only copies the code of the library routines in the executable. Static linking can be easily reverse engineered by pattern matching attacks [114]. Techniques from the field automatic program recognition [131] can be used to identify common programming patterns and replace them with less obvious ones. For example, consider iterating over a linked list; the standard list data structure can be replaced with a less common one, such as cursors into an array of elements.

Attacks: The machine learning based attack of Bichset et al. [22], which was already mentioned as an attack for removing comments and changing formatting, is also applicable for bypassing this transformation.

Modify inheritance relations Programs written in some object-oriented programming languages are distributed in some intermediate format to end-users (e.g. C#, Java, etc.). These intermediate formats are only compiled to native code on the client's machine and contain useful object-oriented programming abstractions. In such programs it is important to break the useful abstractions offered by classes, their structure and their relations (e.g. aggregation, inheritance, etc.). According to Collberg et al. [40], the complexity of a class grows with its depth in the inheritance hierarchy and the number of its direct descendants. This can be done by splitting classes and inserting dummy classes. One variant of class insertion is called *false refactoring* [40]. False refactoring is performed on two or more classes that have no common behavior. All instance variables of these classes having the same type are moved into the new parent class. Methods of the parent class can be buggy versions of methods from its child classes. This approach has been further extended by Foket et al. [60], who propose a technique called *class hierarchy flattening*. In this approach a common interface that contains all methods of all classes is created. All classes implement this common interface and they have no other relationship between each other. This effectively destroys class hierarchies and forces the attacker to analyze the code.

Attacks: At the time of writing this chapter there have been no reported attacks on this obfuscation technique in the literature.

Function argument randomization Randomizing the order of formal parameters of methods and inserting bogus arguments is a technique implemented by tools such as Tigress [38]. The purpose of this transformation is to hide common function signatures across a large diverse set of instances. This transformation is straightforward to perform for programs which do not offer an external interface (e.g. libraries). However, if this obfuscation is applied to a library, then it changes the interface (of that library), and all the corresponding programs using that library will have to be updated as well. The resilience and cost of this transformation are low. However, the resilience can be improved by combining this transformation with the *encode arithmetic* transformation such that computations inside the function are made dependent on the randomly added arguments similarly to how we did for MBA.

Attacks: The static code understanding attack of Zhang et al. [139] is also able to bypass this obfuscation transformations.

4.5 Summary of Survey

In the survey presented above, we have enumerated several practical data and code obfuscation transformations. Practical obfuscation does not offer provable security guarantees like cryptographic obfuscation [17, 62] does. Nevertheless, many contexts mandate the use of practical obfuscation transformations to protect digital software assets, e.g. secret keys, premium content, intellectual property of code, etc. In such contexts, the goal is to raise the bar against the majority of MATE attackers, not all possible attackers, e.g. developers are concerned about malicious end-users, not governmental organizations, which are highly funded.

Table 3 shows an overview and classification of all the obfuscation transformation presented in this section. We note that most of the presented transformations are applicable for

Obfuscation Transformation	**Abstraction**	**Unit**	**Dynamics**	**Target**
Opaque Predicates	All	Function	Static	Data constant
Convert static data to procedural data	All	Instruction	Static	Data constant
Mixed Boolean Arithmetic	All	Basic block	Static	Data constant
White-box cryptography	All	Function	Static	Data constant
One-way transformations	All	Instruction	Static	Data constant
Split variables	All	Function	Static	Data variable
Merge variables	All	Function	Static	Data variable
Restructure arrays	Source	Program	Static	Data variable
Reorder variables	All	Basic block	Static	Data variable
Dataflow flattening	Binary	Program	Static	Data variable
Randomized stack frames	Binary	System	Static	Data variable
Data space randomization	All	Program	Static	Data variable
Instruction reordering	All	Basic block	Static	Code logic
Instruction substitution	All	Instruction	Static	Code logic
Encode Arithmetic	All	Instruction	Static	Code logic
Garbage insertion	All	Basic block	Static	Code logic
Insert dead code	All	Function	Static	Code logic
Adding and removing calls	All	Program	Static	Code logic
Loop transformations	Source, IR	Loop	Static	Code logic
Adding and removing jumps	Binary	Function	Static	Code logic
Program encoding	All	All buy System	Dynamic	Code logic
Self-modifying code	All	Program	Dynamic	Code logic
Virtualization obfuscation	All	Function	Static	Code logic
Control flow flattening	All	Function	Static	Code logic
Branch functions	Binary	Instruction	Static	Code logic
Merging and splitting functions	All	Program	Static	Code abstraction
Remove comments and change formatting	Source	Program	Static	Code abstraction
Scrambling identifier names	Source	Program	Static	Code abstraction
Removing library calls and programming idioms	All	Function	Static	Code abstraction
Modify inheritance relations	Source, IR	Program	Static	Code abstraction
Function argument randomization	All	Function	Static	Code abstraction

Table 3: Overview of the classification of obfuscation transformations

all levels of abstraction (i.e. source code, IR and binary). Most of the presented transformations are applicable at the function unit of transformation, followed closely by the program unit and then by instruction and basic block, both in third place. Only randomized stack frame have an effect on the system unit of transformation, because they change the layout of the stack. Two of the presented techniques are dynamic, meaning that they must allow code pages to be writable during program execution. The number of presented transformations are relatively balanced between the two different targets of transformation and their subcategories, however, code transformations are slightly more numerous.

5 Discussion

In this section we present an overview of the previous survey and discuss the observations stemming from it. Based on this discussion we identify gaps in the field of software obfuscation which require further investigation/research.

Table 4 provides an overview of the various MATE attacks (already mentioned in the previous section), which have been successfully applied in order to defeat the obfuscation transformations presented in section 4, w.r.t. the attack dimensions presented in section 3.

Obfuscation Transformation	Code Understanding				Data item recovery				Location recovery			
	Static		Dynamic		Static		Dynamic		Static		Dynamic	
	Syn	Sem	Syn	Sem	Syn	Sem	Syn	Sem	Syn	Sem	Syn	Sem
Opaque Predicates	[107]	[139]	[107]	[9]		[47]		[9]		[47]		
Convert static data to procedural data	[107]	[139]	[107]	[9]				[9]				
Mixed Boolean Arithmetic						[67]						
White-box cryptography								[48, 134]				
One-way transformations								[117, 129]				
Split variables								[115]				
Merge variables								[115, 126]				[126]
Restructure arrays				[116]								
Reorder variables						[65]						
Dataflow flattening												
Randomized stack frames								[118]				
Data space randomization								[118]				
Instruction reordering		[139]										
Instruction substitution				[9]				[9]				
Encode Arithmetic	[107]		[107]	[9]		[54]		[9]				
Garbage insertion				[136]								
Insert dead code	[107]	[139]	[107]	[9, 136]				[9]				
Adding and removing calls		[139]	[15, 132]									
Loop transformations				[116]								
Adding and removing jumps				[136, 137]								
Program encoding	[107, 121]		[107]									[103]
Self-modifying code		[46]					[96]					[103]
Virtualization obfuscation	[107]	[105]	[11, 107]	[9, 136]		[80]		[9]				
Control flow flattening	[107]		[107]	[9, 123]				[9]				
Branch functions		[83]										
Merging and splitting functions		[106]								[106]		
Remove comments and change formatting	[22]											
Scrambling identifier names	[22]											
Removing library calls and prog. idioms	[22]				[114]							
Modify inheritance relations												
Function argument randomization		[139]										

Table 4: Overview of obfuscation resilience against MATE attacks

These attacks are placed in the cells of Table 4 to indicate the obfuscation transformation(s) they claim to break and where they stand w.r.t. the attack dimensions presented in section 3. The following observations stem from Table 4:

- Some works present attacks which are applicable to multiple obfuscation transformations and multiple attack dimensions. For instance, the attacks by Salem and Banescu [107], Zhang et al. [139], Yadegari et al. [136] and Banescu et al. [9] have a wide range of application. However, not that this does not imply that these attacks are equally effective against all the obfuscation transformation they apply to. For instance symbolic execution based attacks have a much harder task when dealing with *virtualization obfuscation* or *encode arithmetic*, than when dealing with *converting static data to procedural data*.

- Except for comparing different MATE attacks w.r.t. the classification dimensions presented in section 3, there is no set of standard benchmarks used in the field of software obfuscation, which would allow comparing different attacks to one another. This indicates that more research is necessary in order to be able to compare these attack techniques to each other. Such research is needed for evaluating the strength of different obfuscation transformations. Therefore, the development of: (1) standard obfuscation benchmarks for MATE attacks and (2) standard attack benchmarks for obfuscation transformations is still an open problem in the field of software protection.

- It is not shown in Table 4 due to lack of horizontal space, however, there are fewer active attacks, than passive attacks. This may be due to the fact that active attacks are often associated with malicious exploitation (e.g. cracking software), which is not ethical for most researchers. However, more research is required in the direction of active attacks in order to enable the development of stronger protection against such attacks.

- Also there are fewer attacks which aim to recover the location of data and/or code, in comparison to attacks on code understanding or data recovery. Location attacks are often associated with active attacks, therefore the previous argument applies to this observation as well.

- As expected, the majority of data recovery attacks target data obfuscation transformations and the majority of code understanding attacks target code obfuscation transformations. However, there are also some attacks which target both types of transformations, when they are used in combination with each other. Nevertheless, much more research is needed for investigating the strength of combinations of multiple software obfuscation transformations.

- Most attacks on obfuscated code are dynamic, because the majority of the obfuscation transformations presented in this paper aim to break static analysis techniques. Similarly, most attacks are semantic because most dynamic attacks are also semantic. However, note that there are also exceptions. Nevertheless, there are relatively few works about static semantic and dynamic syntactic attacks, which indicates a topic that needs further exploration.

- Obfuscation transformations which have few or no attacks against them are not an indication that they are more secure. For instance, in the case of *dataflow flattening*, the most probable reason why there are no attacks against it is because it is too expensive to implement this obfuscation transformation in most practical scenarios. One of the strongest obfuscation transformation categories against MATE attackers are dynamic transformations, however, these transformations generally require making executable code pages in process memory also writable, which opens the door to remote code injection attacks [125]. Since most software today is connected to the Internet, the risk of a remote attacker performing code injection in the software of an end-user is higher than the risk of that end-user being a MATE attacker. Therefore, such self-modifying code techniques are avoided by commercial software developers.

- Most attacks published in literature are dynamic, because even if code is statically encrypted/encoded, it must be decrypted/decoded during execution. This is the "Achilles heel" of obfuscation, i.e. code must still be executable, no matter how intricate the code is obfuscated when inspected statically. To prevent dynamic MATE attacks in practice obfuscation is used in combination with anti-dynamic-analysis techniques such as anti-debugging [1] and anti-emulation [57]. Such methods aim to prevent dynamic analysis attacks, i.e. they force attackers to take a static analysis approach, because obfuscation is assumed to be harder to break using static attacks.

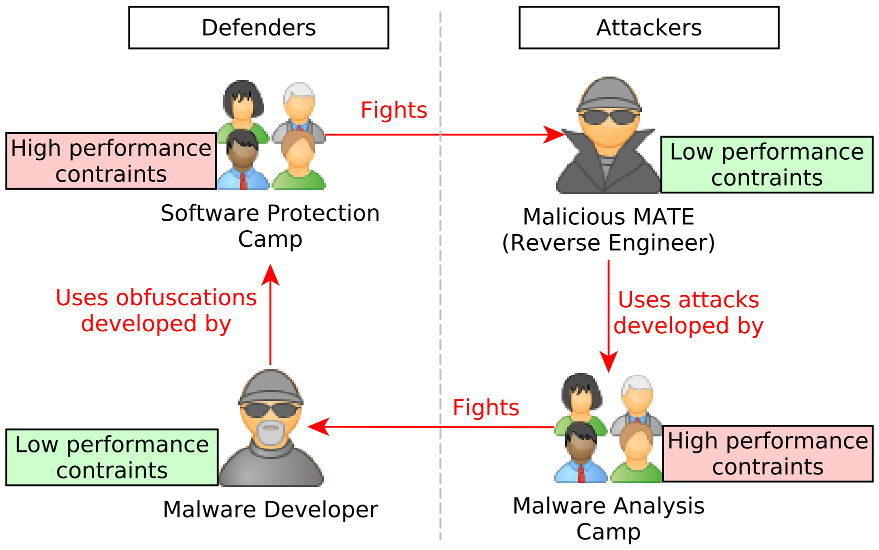
Defenders
Attackers
Fights
High performance contraints
Software Protection Camp
Low performance contraints
Malicious MATE (Reverse Engineer)
Uses obfuscations developed by
Uses attacks developed by
Fights
Low performance contraints
Malware Developer
High performance contraints
Malware Analysis Camp

Figure 4: Overview of obfuscation related research camps and their connection via malicious entities.

- In some contexts, the MATE attacker is not forced to perform static analysis due to anti-dynamic-analysis techniques, but due to resource constraints. For instance, some malware analysis systems have to analyze millions of instances per day [95]. Performing dynamic analysis (e.g. execution trance analysis) is in general more costly than static analysis (e.g. code pattern recognition). Therefore, benign MATE attackers (i.e. malware analysts) choose to perform static analysis attacks as a way of quickly filtering previously analyzed malware instances, which employ simple obfuscation transformations such as *instruction reordering*. Generally, any software which is detected as highly obfuscated and not seen before is subjected to a more thorough (dynamic) analysis.

Given the publications in the field of software obfuscation presented in this chapter, we can see that the field of software protection research is divided into two camps:

1. *The software protection camp*, which aims to create defenses against malicious MATE attackers who want steal intellectual property and bypass license or integrity checks. The main disadvantage of this camp is that it is subject to significant performance constraints, because commercial software developers generally do not want to compromise the speed or responsiveness of their products by adding security.

2. *The malware analysis camp*, which are benign MATE attackers, who aim to develop attacks against obfuscation techniques employed by malware. The main disadvantage of this camp is again related to performance constraints of their attacks, which should be automatic and scalable to millions of programs per day.

These constraints are due to the high competitiveness on the commercial software markets. In today's software market the end-user experience is one of the main selling points of software and this implies having transparent but effective security solutions.

Probably the main beneficiaries of the publications from this field are malicious MATE attackers and malware developers (shown in the top-right and bottom-left of Figure 4) who can borrow any techniques proposed by the second and first camps, respectively, because they do not impose such high performance constraints on themselves, as those imposed on the two camps of researchers. For instance, a malware developer is likely much more interested in the security added by combining multiple obfuscation transformations, because even if the performance of the malware is decreased by three orders of magnitude, for the malware developer it is crucial that malware analysis engines have a hard-time disarming and reverse engineering the malware. This is very different for game developers, who must have highly responsive graphical user interfaces. Similarly to malware developers, malicious reverse engineers may afford to leave their laptops to run an attack for "a few more hours/-days", because the return on investment (e.g. a free, cracked game) is likely higher than the power bill.

Given this observation, one may jump to the conclusion that it may be beneficial for benign software developers and anti-malware developers, if less research in this field were published, because then malicious MATE attackers and malware developers would not have as many techniques and tools at their disposal. This would not stop malicious parties from developing their own techniques, but it would keep many benign parties "in the dark" about advances in this field. This was the case in the 1980s, before this research community was born and malicious parties were already using software obfuscation to protect malware and MATE attacks to break licenses. Therefore, having such a two camp research community, where each side challenges the weaknesses of the other side leads to continuous improvements, which accelerate progress and practical applicability of such research.

6 Related Work

As indicated in Figure 1, there are several ways in which one could protect against MATE attacks. Since in this chapter we focus on technical protection via obfuscation, in this section we present the other types of protections. This is followed by related work from the field of cryptographic obfuscation and by other surveys of software obfuscation.

6.1 Encryption via Trusted Hardware

Software protection via encryption is usually enabled by trusted hardware, also called *trusted computing*. Intel has released a hardware based technology [3], known as *Software Guard eXtension* (SGX), which enables software developers to protect the confidentiality of their applications' code via protected execution areas called *enclaves*. Dewan et al. [50] also use a trusted hypervisor to protect the sensitive memory of programs against unauthorized access by leveraging trusted hardware. Feng et al. [56] propose performing randomly-timed stealthy measurements which can be validated locally, using Intel's Active Management Technology [73]. These approaches provide high security guarantees. However, they require trusted hardware to be available and the installation of a hypervisor. Software developers of popular software (e.g. web browsers), generally do not want to restrict their user base by imposing such requirements.

6.2 Server-Side Execution

Tamper protection via communication with trusted servers is employed in massive multiplayer online games (MMOGs) to detect cheating. Anti-cheat software such as PunkBuster [53], Valve Anti-Cheat (VAC) [124], Fides [77] and Warden [69] perform client-side computation, which are validated by a trusted server.

Martignoni et al. [91] and Seshandri et al. [111] propose establishing a *trusted computing base* to achieve verifiable code execution on a remote un-trusted system. The trusted computing base in the two methods is established using a verification function. The verification function is composed of three components: (i) a checksum function, (ii) a send function, and (iii) a checksum function. However, the main difference between the two methods is the checksum function. In the work of Martignoni et al. [91] generates a new checksum function each time and sends it encrypted to the un-trusted system. In the work of Seshandri et al. [111], the checksum function is known a priori and the challenge issued by the dispatcher consists in a seed that initializes this function. Since the remote component in both methods knows precisely in which execution environment the function must be executed and knows the hardware characteristics of the un-trusted system, it can compute the expected checksum value and can estimate the amount of time that will be required by the un-trusted system to decrypt and execute the function, and to send back the result. Since Intel x86 architecture, the architecture for which the approach of Seshandri et al. [111], was developed, is full of subtle details, researchers have found ways to circumvent the remote component. Also, a limitation of the approach of Martignoni et al. [91], is the impossibility to bootstrap a tamper-proof environment on simultaneous multi threading (SMT) or simultaneous multi processing (SMP) systems. On such systems, the attacker can use the secondary computational resources (parallel threads for example) to forge checksums or to regain control of the execution after attestation.

Jakobsson and Johansson [76] propose a similar technique for detecting malware on mobile devices. Collberg et al. [38] propose tamper protection by pushing continuous updates from a trusted server to the client, which force the attacker to repeat reverse engineering and patching on each update. One disadvantage of these protection techniques is their dependence on external trusted servers. This dependence may cause a denial-of-service to end-users of the protected software applications which are also meant to be used offline, in case Internet connectivity is unavailable. Therefore in this chapter we focus on solutions that operate locally, i.e., without dependence on a trusted server.

6.3 Code Tamper-detection and Tamper-proofing

Code tamper-detection and tamper-proofing are complementary techniques to software diversity and obfuscation and they aim to detect, respectively prevent unauthorized modifications of a program's code. However, these techniques are not generally stealthy, and hence they should be combined with diverse obfuscation in order to hamper MATE attackers from disabling such mechanisms.

Chang and Atallah [30] propose building a network of code regions, where a region can be a block of user code, a checker, or a responder. In this method checkers check each other in addition to user code by comparing a known checksum of piece of code to runtime checksum of the same code. If the checker has discovered that a region has been tampered with, a responder will replace the tampered region with a copy stored elsewhere. An important aspect of this algorithm is that it is not enough for checkers to check just the code, they must check each other as well. If checkers are not checked, they are easy to remove. Horne et al. [70] build on top of [30], by hiding the expected (precomputed checksum) value which is easy to identify, because of its randomness. The idea is to construct the checksum function such that unless the code has been tampered with, the function always checksums to a known number (usually zero). Having this function allows to insert an empty slot within the region under protection, and later give this slot a value that makes the region checksum to zero. The technique of Horne et al. [70] randomly places large numbers of checkers all over the program, but makes sure that every region of code is covered by multiple checkers. To minimize pattern-matching attacks, this method describes how to generate a large number of variants of lightweight checksum functions. The disadvantage of the *code introspection* approach used by both [30] and [70] is its stealthiness, because code that reads itself is seldom used for other purposes.

Chen et al. [31] propose an idea called *oblivious hashing*, where the checksum value is computed over the *execution trace* rather than the static code. The checksum can be computed by inserting instructions that monitor changes to variables and the execution of instructions. A problem with automating this technique, is that it is hard to predict what side effects a function might have. It might destroy valuable global data or allocate extraneous dynamic memory that will never be properly freed. Furthermore, there is a problem with non-deterministic functions that depend on: user input, the time of day, network traffic, thread scheduling, and so on, because they do not have a fixed output that can be checked. This technique also faces the issue of automatically generating challenge data (test inputs) that most of the code of a function. The approach by Ibrahim and Banescu [72] implements a variant of oblivious hashing and therefore it also suffers from the same disadvantages. However, we address the last issue by proposing the use of symbolic execution in order to generate the challenge data.

Jacob et al. [75] propose an approach which depends on a unique property of the x86 instruction set architecture (ISA). The x86 ISA has a *variable instruction length* (1-15 bytes) with no alignment, this means instructions can start at any offset in the code. This results in the possibility of having overlapping or even nested instructions. So the basic idea will be that when a block is executed it computes a checksum of another block. For the purpose of protecting the code, we need two blocks to share instruction bytes. Having two blocks to share instruction bytes, can be achieved by interleaving the instructions and inserting jumps to maintain semantics. The advantage in this technique is that the code checksumming computations will not require reading the code explicitly. The disadvantage is mainly the performance overhead of the added instructions. Jacob et al. [75] report that the protected binary can be up to three times slower than the original. Even though this overhead may be acceptable in many circumstances, this technique cannot be applied to programs that execute on the Common Language Runtime such as programs written in C#.

Cappaert et al. [28] propose a technique that hinders both code analysis and tampering attacks simultaneously through code encryption. During run-time, code decryption can be done at a chosen granularity (e.g. one function at a time), when that part of code is needed at run-time. This technique performs integrity-checking of the code by using it to compute the keys for decryption and encryption. The basic idea is using the checksum value of a function, as the decryption key of another function. The advantage of this technique is that the encryption key is computed at run time, which means the key is not hard-coded in the binary and therefore hard to find through static analysis. The disadvantage of this technique is the run-time overhead as well as the its stealth.

6.4 Cryptographic Obfuscation

In addition to the practical software protection techniques presented so far, there is also an entire sub-field of cryptography dedicated to obfuscation. The first formal study of obfuscation was published in 2001 by Barak et al. [17]. They propose that an ideal obfuscator should be able to take any program and transform it into a *virtual black box*, i.e. a MATE attacker would be able to interact with it in the same manner as with a program running on a remote server, however, the attacker would not be able to learn anything from the program in addition to what can be learned from its input-output behavior. Barak et al. [17] formally prove that such an obfuscator cannot exist for all programs. However, they do not exclude that such an obfuscator may exist for particular programs.

Over a decade later, Garg et al. [62] proposed a construction for *indistinguishability obfuscation*, a different obfuscation notion than black-box obfuscation, which guarantees that the obfuscations of two programs implementing the same functionality are computationally indistinguishable. This was a major breakthrough in cryptography, since a few years earlier it was proven by Goldwasser and Rothblum [64] that indistinguishability obfuscation is the best possible type of obfuscation that can be achieved for all programs. Therefore, we are currently seeing a revival of interest in obfuscation from the cryptographic community, because the construction of Garg et al. [62] may be employed to construct functional encryption, public key encryption, digital signatures, etc. However, multiple works have signaled that the current constructions of indistinguishability obfuscation are still far from being practical, i.e. applicable to real-world software applications [6, 13, 16].

6.5 Other Surveys of Software Obfuscation

The seminal work of Collberg et al. [40] provides one of the first taxonomies of software obfuscation. Their work also provides an illustration of how different obfuscation transformations work, however since this paper was published 20 years before this chapter, they do not include recently developed obfuscation transformations and attacks against them, as presented in this chapter. Mavrogiannopoulos et al. [94] provide a taxonomy of self-modifying code techniques and implementations, which we have compressed into one single transformation.

Larsen et al. [86] provide a survey of software diversification transformations, which are closely related to obfuscation, i.e. all software obfuscation transformations can be used to achieve diversity, but not vice-versa. However, as opposed to our work which focuses on MATE attacks, the work of Larsen et al. [86] focuses on remote attacks such as memory corruption, code injection and code reuse.

Schrittwiesser et al. [109] recently published a survey complementary to this work. Their focus is on providing an overview of MATE attacks and how they affect existing obfuscation transformations. However, they do not provide many details or illustrations of how the different obfuscation transformations look like, as we do in this chapter. Their classification of obfuscation transformations and MATE attacks differ from the classification used in this chapter to some extent. Therefore, they do not touch on the same points we have discussed in section 5. For instance, as opposed to our work, they do not discuss obfuscation transformations that target the readability of the source code, hence, they also do not cite MATE attacks applicable to such transformations. Regarding the MATE attack classification, we emphasize the difference between syntactic and semantic attacks and claim that this dimension of classification is orthogonal to the dynamics dimension. This means that syntactic attacks such as pattern recognition can also be applied to dynamically generated execution traces, which is often done by malware analysis engines. Moreover, we also propose using the *alternation* classification dimension, which differentiates between active and passive attacks. This dimension allowed us to identify the fact that there are far less active attacks published in the literature than passive attacks, which indicates a gap in this field. On the other hand, they propose a separation between automated attacks and human-assisted attacks. They claim human-assisted attacks are the hardest attacks to defeat using obfuscation.

7 Conclusion

This paper presents a tutorial on software obfuscation. This tutorial includes a state of the art survey of obfuscation transformations and their classifications. Each obfuscation transformation is accompanied by an illustrative example (often in the form of code snippets), as well as an enumeration of existing associated attacks published in the literature. We discuss how the current landscape of software obfuscation research is split into two complementary camps, namely software protection and malware analysis, and why this division is important for accelerating progress. An interesting observation is that malicious entities may have higher benefits from techniques proposed in the field of research, due to the fact that they do not have high performance constraints as benign entities (e.g. commercial software vendors) have. However, the only way to defend against malicious entities is to push these areas of research forward. One way to increase the performance constraints of malicious MATE attackers is by employing frequent software updates, which shrinks the window of opportunity of such attackers. Therefore, future research in the field of software obfuscation should also investigate obfuscation techniques which allow incremental updates.

Our outlook on the field of obfuscation is positive. We believe that even with advances in hardware-based software protection techniques such as Intel SGX [42], software obfuscation will still be used for applications where such hardware is not available (e.g. mobile devices), or where such hardware is too costly too incorporate (e.g. resource constrained embedded devices). Moreover, Intel SGX may be able to defend software assets against a software-based attacks, but MATE attackers may also resort to side-channel attacks [25] or reverse engineering attacks at the hardware level [97]. There is an entire field of research, parallel to software obfuscation, namely *hardware obfuscation*, where the goal is to mangle the structure or layout of logical gates such that MATE attackers cannot steal the intellectual property directly from hardware, e.g. an Application-Specific Integrated Circuit (ASIC) [20]. Note that it is straightforward to map many of the obfuscation transformations presented in this chapter to the field of hardware obfuscation, and vice-versa. Furthermore, the field of cryptograpic obfuscation is popular nowadays due to the indistinguishability obfuscation candidate proposed by Garg el al. [62], which makes us optimistic in believing that obfuscation will still be a highly dynamic and innovative field of research in the following decade.

The main challenge in the field of software obfuscation is that there is no standard methodology or benchmarks for evaluating the strength of different obfuscation transformations or combinations thereof. The first steps in the direction of obfuscation evaluation have been made by Banescu et al. [9, 10, 12], who use automated attack effort as a way of comparing the strength of different obfuscation transformations. However, they mainly focus on attacks based on symbolic execution, hence more research is needed in this direction to cover other types of attacks. This observation is on par with the conclusions of other surveys [109]. We also note that more research is required for topics such as the development of active attacks, static semantic attacks and self-modifying code techniques that do not increase the attack surface of the software they protect.

We envision that side-channel attacks against obfuscation will become increasingly popular. This premonition is due to the recent *Differential Computation Analysis* (DCA) presented by Bos et al. [24]. DCA is the software counterpart of *Differential Power Analysis* (DPA), which has been succesfully applied to recover secret keys from ASICs, e.g. smart-cards with little effort and prerequisites [82]. Similar to DPA, DCA able to recover a symmetric cryptographic key from a white-box cryptographic cypher binary, in a matter of seconds, without needing to disassemble the binary or to know anything about its structure. However, these side-channel attacks also make some assumptions, which could be broken by cleaver obfuscation transformations. Therefore, we urge researchers to invest more resources in designing obfuscation transformations, which are able to block such side-channel attacks.

References

[1] B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, and B. De Sutter. Tightly-coupled self-debugging software protection. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, page 7. ACM, 2016.

[2] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. K. Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44–57, 2015.

[3] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[4] B. Anckaert, M. H. Jakubowski, R. Venkatesan, and C. W. Saw. Runtime protection via dataflow flattening. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 242–248. IEEE, 2009.

[5] L. O. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, University of Cophenhagen, 1994.

[6] D. Apon, Y. Huang, J. Katz, and A. J. Malozemoff. Implementing cryptographic program obfuscation. *IACR Cryptology ePrint Archive*, 2014:779, 2014.

[7] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[8] L. Badger, L. D'Anna, D. Kilpatrick, B. Matt, A. Reisse, and T. Van Vleck. Self-protecting mobile agents obfuscation techniques evaluation report. *Network Associates Laboratories, Report*, pages 01–036, 2002.

[9] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In *Proc. of 2016 Annual Computer Security Applications Conference*. ACM, 2016.

[10] S. Banescu, C. Collberg, and A. Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017. USENIX Association.

[11] S. Banescu, C. Lucaci, B. Krämer, and A. Pretschner. Vot4cs: A virtualization obfuscation tool for c. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 39–49. ACM, 2016.

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

[13] S. Banescu, M. Ochoa, A. Pretschner, and N. Kunze. Benchmarking indistinguishability obfuscation - a candidate implementation. In *Proc. of 7th International Symposium on ESSoS*, number 8978 in LNCS, 2015.

[14] S. Banescu, A. Pretschner, D. Battré, S. Cazzulani, R. Shield, and G. Thompson. Software-based protection against changeware. In *Proceedings of the Conference on Data and Application Security and Privacy*, CODASPY '15, pages 231–242, 2015.

[15] S. Banescu, T. Wuchner, A. Salem, M. Guggenmos, A. Pretschner, et al. A framework for empirical evaluation of malware detection resilience against behavior obfuscation. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 40–47. IEEE, 2015.

[16] B. Barak. Hopes, fears, and software obfuscation. *Communications of the ACM*, 59(3):88–96, 2016.

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

[18] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289. ACM, 2003.

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

[20] G. T. Becker, M. Fyrbiak, and C. Kison. Hardware obfuscation: Techniques and open challenges. In *Foundations of Hardware IP Protection*, pages 105–123. Springer, 2017.

[21] S. Bhatkar and R. Sekar. Data space randomization. In D. Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, number 5137 in Lecture Notes in Computer Science, pages 1–22. Springer Berlin Heidelberg, Jan. 2008.

[22] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355. ACM, 2016.

[23] O. Billet, H. Gilbert, and C. E. Chatbi. Cryptanalysis of a white box AES implementation. SAC'04, pages 227–240, Waterloo, Canada, 2005. Springer-Verlag.

[24] J. W. Bos, C. Hubain, W. Michiels, and P. Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 215–236. Springer, 2016.

[25] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521*, 2017.

[26] J. Bringer, H. Chabanne, and E. Dottax. White box cryptography: Another attempt. *located at, last visited on Jul*, 22(2011):14, 2006.

[27] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical report, Technical Report TR-2008-120, Microsoft Research, 2008. Cited on, 2008.

[28] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. De Bosschere. Towards tamper resistant code encryption: Practice and experience. In *Information Security Practice and Experience*, pages 86–100. Springer, 2008.

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

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

[31] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *International Workshop on Information Hiding*, pages 400–414. Springer, 2002.

[32] S. Chow, P. Eisen, H. Johnson, and P. C. V. Oorschot. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, number 2595 in LNCS, pages 250–270. Springer Berlin Heidelberg, Jan. 2003.

[33] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot. A white-box DES implementation for DRM applications. In *Digital Rights Management*, pages 1–15. Springer, 2003.

[34] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *International Conference on Information Security*, pages 144–155. Springer, 2001.

[35] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, Oct. 1993.

[36] C. Collberg. The Tigress C Diversifier/Obfuscator. `http://tigress.cs.arizona.edu/`. Accessed: 2016-11-29.

[37] C. Collberg, J. Davidson, R. Giacobazzi, Y. X. Gu, A. Herzberg, and F. Wang. Toward digital asset protection. *Intelligent Systems, IEEE*, 26(6):8–13, 2011.

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

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

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

[41] U. Congress. Digital millennium copyright act. *Public Law*, 105(304):112, 1998.

[42] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[43] N. T. Courtois and J. Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 267–287. Springer, 2002.

[44] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[45] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[46] M. Dalla Preda, R. Giacobazzi, S. Debray, K. Coogan, and G. M. Townsend. Modelling metamorphism by abstract interpretation. In *International Static Analysis Symposium*, pages 218–235. Springer, 2010.

[47] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. Opaque predicates detection by abstract interpretation. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology*, volume 4019 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin Heidelberg, 2006.

[48] Y. De Mulder, P. Roelse, and B. Preneel. Cryptanalysis of the Xiao-Lai white-box AES implementation. In *Selected Areas in Cryptography*, pages 34–49, 2013.

[49] B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. De Bosschere. Instruction set limitation in support of software diversity. In *International Conference on Information Security and Cryptology*, pages 152–165. Springer, 2008.

[50] P. Dewan, D. Durham, H. Khosravi, M. Long, and G. Nagabhushan. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proceedings of the 2008 Spring simulation multiconference*, pages 828–835. Society for Computer Simulation International, 2008.

[51] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, SFCS '81, pages 350–357, Washington, DC, USA, 1981. IEEE Computer Society.

[52] R. El-Khalil and A. D. Keromytis. Hydan: Hiding information in program binaries. In *International Conference on Information and Communications Security*, pages 187–199. Springer, 2004.

[53] Evenbalance. PunkBuster — Online Countermeasures, 2015. `http://www.evenbalance.com/pbsetup.php`, [Online; accessed 20-September-2016].

[54] N. Eyrolles, L. Goubin, and M. Videau. Defeating mba-based obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 27–38. ACM, 2016.

[55] R. Fedler, S. Banescu, and A. Pretschner. Isa2r: Improving software attack and analysis resilience via compiler-level software diversity. In *International Conference on Computer Safety, Reliability, and Security*, pages 362–371. Springer, 2015.

[56] W.-c. Feng, E. Kaiser, and T. Schluessler. Stealth measurements for cheat detection in on-line games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 15–20, 2008.

[57] P. Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 55, 2007.

[58] P. FIPS. 197: Advanced encryption standard (aes). *National Institute of Standards and Technology*, 2001.

[59] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[60] C. Foket, B. De Sutter, B. Coppens, and K. De Bosschere. A novel obfuscation: class hierarchy flattening. In *Foundations and Practice of Security*, pages 194–210. Springer, 2013.

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

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

[63] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[64] S. Goldwasser and G. N. Rothblum. On best-possible obfuscation. In *Theory of Cryptography*, pages 194–213. Springer, 2007.

[65] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 101–120. Springer, 2009.

[66] GuardSquare. ProGuard: The open source optimizer for Java bytecode. `https://www.guardsquare.com/en/proguard`. Accessed: 2017-03-03.

[67] A. Guinet, N. Eyrolles, and M. Videau. Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions. In *GreHack 2016*, 2016.

[68] R. Gupta, D. Benson, and J. Z. Fang. Path profile guided partial dead code elimination using predication. In *Parallel Architectures and Compilation Techniques., 1997. Proceedings., 1997 International Conference on*, pages 102–113. IEEE, 1997.

[69] G. Hoglund. Hacking world of warcraft: An exercise in advanced rootkit design. *Black Hat*, 2006.

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

[71] S. Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):1–6, 1997.

[72] A. Ibrahim and S. Banescu. Stins4cs: A state inspection tool for c. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 61–71. ACM, 2016.

[73] Intel. Intel Active Management Technology — Query, Restore, Upgrade, and Protect Devices Remotely, 2016. `http://www.intel.com/content/www/us/en/architecture-and-technology/intel-active-management-technology.html`, [Online; accessed 20-September-2016].

[74] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan. The superdiversifier: Peephole individualization for software protection. In *Advances in Information and Computer Security*, pages 100–120. Springer, 2008.

[75] M. Jacob, M. H. Jakubowski, and R. Venkatesan. Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *Proceedings of the 9th workshop on Multimedia & security*, pages 129–140. ACM, 2007.

[76] M. Jakobsson and K.-A. Johansson. Retroactive detection of malware with applications to mobile platforms. In *Proceedings of the 5th USENIX Conference on Hot Topics in Security*, HotSec'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[77] E. Kaiser, W.-c. Feng, and T. Schluessler. Fides: Remote anomaly-based cheat detection using client emulation. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 269–279, New York, NY, USA, 2009. ACM.

[78] Y. Kanzaki, A. Monden, M. Nakamura, and K.-i. Matsumoto. Exploiting self-modification mechanism for program protection. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 170–179, 2003.

[79] M. Karroumi. Protecting white-box AES with dual ciphers. In K.-H. Rhee and D. Nyang, editors, *Information Security and Cryptology - ICISC 2010*, number 6829 in Lecture Notes in Computer Science, pages 278–291. Springer Berlin Heidelberg, Jan. 2011.

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

[81] J. Knoop, O. Rüthing, and B. Steffen. *Partial dead code elimination*, volume 29. ACM, 1994.

[82] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in cryptology—CRYPTO'99*, pages 789–789. Springer, 1999.

[83] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.

[84] A. Kur, M. Planck, and T. Dreier. *European intellectual property law: text, cases and materials.* Edward Elgar Publishing, 2013.

[85] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[86] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014.

[87] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[88] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In *Information Security Applications*, pages 194–206. Springer, 2006.

[89] A. Main and P. C. van Oorschot. Software protection and application security: Understanding the battleground. *International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography,*, 2003.

[90] A. Majumdar and C. Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, pages 187–196. Australian Computer Society, Inc., 2006.

[91] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40. Springer, 2010.

[92] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 524–533. ACM, 2009.

[93] J. Mason, K. Watkins, J. Eisner, and A. Stubblefield. A natural language approach to automated cryptanalysis of two-time pads. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 235–244. ACM, 2006.

[94] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, Nov. 2011.

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

[96] A. Nguyen-Tuong, A. Wang, J. D. Hiser, J. C. Knight, and J. W. Davidson. On the effectiveness of the metamorphic shield. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 170–174. ACM, 2010.

[97] K. Nohl, D. Evans, S. Starbug, and H. Plötz. Reverse-engineering a cryptographic rfid tag. In *USENIX security symposium*, volume 28, 2008.

[98] M. Oberhumer, L. Molnár, and J. F. Reiser. Upx: the ultimate packer for executables, 2004.

[99] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, pages 308–316. IEEE, 2000.

[100] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. pages 601–615. IEEE, May 2012.

[101] PreEmptiveSolutions. DashO: Java & Android Obfuscator & Runtime Protection. `https://www.preemptive.com/products/dasho`. Accessed: 2017-03-03.

[102] PreEmptiveSolutions. Dotfuscator: .NET App Self Protection and Obfuscation. `https://www.preemptive.com/products/dotfuscator`. Accessed: 2017-03-03.

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

[104] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

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

[106] S. Rugaber, K. Stirewalt, and L. M. Wills. The interleaving problem in program understanding. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 166–175. IEEE, 1995.

[107] A. Salem and S. Banescu. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Software Security, Protection and Reverse Engineering Workshop*, page 8. ACM, 2016.

[108] S. Schrittwieser and S. Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding*, pages 270–284, 2011.

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

[110] SemanticDesigns. Thicket Family of Source Code Obfuscators. `http://www.semanticdesigns.com/Products/Obfuscators/`. Accessed: 2017-03-03.

[111] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 1–16. ACM, 2005.

[112] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[113] A. Shamir and N. Van Someren. Playing 'hide and seek' with stored keys. In *Financial cryptography*, pages 118–124, 1999.

[114] I. Skochinsky. IDA F.L.I.R.T. Technology: In-Depth. `https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml`, 2013. Accessed: 2017-03-03.

[115] A. Slowinska, I. Haller, A. Bacs, S. Baranga, and H. Bos. Data structure archaeology: scrape away the dirt and glue back the pieces! In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–20. Springer, 2014.

[116] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[117] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. *URL: https://shattered. it/static/shattered. pdf*, 2017.

[118] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8. ACM, 2009.

[119] Stunnix. C/C++ Obfuscator. `http://stunnix.com/prod/cxxo/`. Accessed: 2017-03-03.

[120] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery.* Pearson Education, 2007.

[121] Y. Tang and S. Chen. An automated signature-based approach against polymorphic internet worms. *IEEE Transactions on Parallel and Distributed Systems*, 18(7), 2007.

[122] P. Team. Pax non-executable pages design & implementation. *Avaliable: http://pax. grsecurity. net*, 2003.

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

[124] Valve. Valve Anti-Cheat System (VAC), 2015. `https://support.steampowered.com/kb_article.php?p_faqid=370`, [Online; accessed 20-September-2016].

[125] A. van de Ven and I. Molnar. Exec shield, 2004.

[126] A. Viticchié, L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella, and R. Tiella. Assessment of source code obfuscation techniques. 2016.

[127] Z. Vrba. cryptexec: Next-generation runtime binary encryption using on-demand function extraction. 2003.

[128] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *International Conference on Dependable Systems and Networks, 2001. DSN 2001*, pages 193–202, 2001.

[129] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions md4, md5, haval-128 and ripemd. *IACR Cryptology ePrint Archive*, 2004:199, 2004.

[130] H. S. Warren. *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[131] L. M. Wills. Automated program recognition: a feasibility demonstration. *Artif. Intell.*, 45(1-2):113–171, Sept. 1990.

[132] T. Wüchner, M. Ochoa, and A. Pretschner. Robust and effective malware detection through quantitative data flow graph metrics. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 98–118. Springer, 2015.

[133] B. Wyseur. *White-Box Cryptography.* PhD thesis, KATHOLIEKE UNIVERSITEIT LEUVEN, Kasteelpark Arenberg 10, 3001 Leuven-Heverlee, 2009.

[134] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In *Selected Areas in Cryptography*, number 4876 in LNCS, pages 264–277. Springer Berlin Heidelberg, 2007.

[135] Y. Xiao and X. Lai. A secure implementation of white-box AES. In *2nd International Conference on Computer Science and its Applications, 2009. CSA '09*, pages 1–6, 2009.

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

[137] B. Yadegari, J. Stephens, and S. Debray. Analysis of exception-based control transfers. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 205–216. ACM, 2017.

[138] yWorks. yGuard Java Bytecode Obfuscator and Shrinker. `https://www.yworks.com/products/yguard`. Accessed: 2017-03-03.

[139] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36. ACM, 2014.

[140] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *International Workshop on Information Security Applications*, pages 61–75. Springer, 2007.