Xu *et al. Cybersecurity* (2020) 3:9
https://doi.org/10.1186/s42400-020-00049-3

Cybersecurity

REVIEW Open Access

Layered obfuscation: a taxonomy of software obfuscation techniques for layered security

Hui Xu[1*], Yangfan Zhou[2], Jiang Ming[3] and Michael Lyu[4]

Abstract

Software obfuscation has been developed for over 30 years. A problem always confusing the communities is what security strength the technique can achieve. Nowadays, this problem becomes even harder as the software economy becomes more diversified. Inspired by the classic idea of layered security for risk management, we propose *layered obfuscation* as a promising way to realize reliable software obfuscation. Our concept is based on the fact that real-world software is usually complicated. Merely applying one or several obfuscation approaches in an ad-hoc way cannot achieve good obscurity. Layered obfuscation, on the other hand, aims to mitigate the risks of reverse software engineering by integrating different obfuscation techniques as a whole solution. In the paper, we conduct a systematic review of existing obfuscation techniques based on the idea of layered obfuscation and develop a novel taxonomy of obfuscation techniques. Following our taxonomy hierarchy, the obfuscation strategies under different branches are orthogonal to each other. In this way, it can assist developers in choosing obfuscation techniques and designing layered obfuscation solutions based on their specific requirements.

Keywords: Software obfuscation, Layered security, Element-layer obfuscation, Component-layer obfuscation, Inter-component obfuscation, Application-layer obfuscation

Introduction

Sofware obfuscation transforms computer programs to new versions which are semantically equivalent with the original ones but much harder to understand (Collberg et al. 1997). It is a technique which protects software intellectual properties against MATE (Man-At-The-End) attacks (Collberg et al. 2011). The concept was originally introduced at the International Obfuscated C Code Contest in 1984, which awarded creative C source codes with "smelly styles". Later in 1997, Collberg et al. (Collberg et al. 1997) published a milestone paper discussing the taxonomy of obfuscation transformations for Java programs. Since then, the technique has become indispensable for software protection. There are many practical obfuscation approaches developed, such as lexical obfuscation with ProGuard (ProGuard 2016) and control-flow obfuscation with Obfuscator-LLVM (Junod et al. 2015).

Critical challenge of obfuscation

Although obfuscation has been developed for over 30 years, the questions yet unsolved are how much developers can trust the technique and how to design reliable obfuscation solutions. Such issues are very critical because obfuscation is a security primitive. To tackle these questions, we have surveyed the literature of both theoretical and practical obfuscation research.

From the theoretical perspective, many discussions (e.g., (Barak et al. 2001; Garg et al. 2013a; Lewi et al. 2016; Zimmerman 2015) on this problem have arisen in recent years. The representative ones include the negative result showed by Barak et al. (2001) that we can-

*Correspondence: xuh@fudan.edu.cn
[1]School of Computer Science, Fudan University, Shanghai, China
Full list of author information is available at the end of the article

not obfuscate all program with black-box security, and the positive result presented by Garg et al. (2013a) that graded encoding is a promising obfuscation algorithm for achieving a weaker security notion: indistinguishability. However, we cannot apply these results to practical software obfuscation directly because there are obvious gaps in between. Note that such theoretical research focuses on obfuscating computation models (e.g., circuits or Turing Machines) instead of real codes. While computation models are mathematical and their properties are usually provable, real codes are more complicated and their properties are hard to prove. In practice, we generally program software with high-level programming languages which cannot be reduced to pure mathematical representations easily.

From the practical area, we attempt to find some clues for designing reliable obfuscation solutions. We find that present obfuscation research generally assumes a specific code format (e.g., Java bytecodes or assembly codes) for obfuscation. However, real-world software can be more complicated than that. For instance, an Android app (Fig. 1) contains several different components, such as Java codes, native codes, third-party libraries, and other resources. Securely obfuscating the whole app with only one approach is nearly impossible. Moreover, merely applying some obfuscation techniques in an ad-hoc way can achieve very limited obscurity because it lacks a holistic design. In particular, the remaining unobfuscated information could jeopardize the obfuscated software. For instance, the lexical obfuscation approach provided by ProGuard (2016) transforms identifiers of Android apps to meaningless alphabets or strings, which seems one-way secure. But a recent attack (Bichsel et al. 2016) shows that attackers can recover a significant portion of the original lexical information leveraging the residual information within the obfuscated apps.

We conjecture that achieving reliable obfuscation is challenging mainly due to the complicated nature of software, and we believe a promising way to handle the challenge is applying the classic idea of *layered security* to software obfuscation.

Layered security for obfuscation

Layered security is an effective risk management strategy. It mitigates the risks that a threat becomes a reality with several protections from different layers or of various types. The idea has become prevalent for securing information systems after it has been introduced by the Department of Defense in Information Assurance Technical Framework (IATF) (2002). Because information systems are very complicated, there is no silver bullet for avoiding all risks, and layered security is the best practice. In the first level, IATF divides information systems into four areas or layers, which are local computing environment, enclave boundaries, network and infrastructures, and supporting infrastructures. Each of these layers faces a specific group of threats and should be protected correspondingly. Take the area of network and infrastructure as an example, administrators can employ firewalls to deter denial-of-service attacks from the internet, and they can use the SSL/TLS (Secure Socket Layer/Transport Layer Security) gateways (Rescorla 2001) to encrypt the traffics from being eavesdropped. The layered security idea integrates different security mechanisms as a whole to protect the security of a system.

Although the software is not as complicated as information systems, its complexity is beyond the capability of any single obfuscation technique. Therefore, we believe employing the idea of layered security for software obfuscation should be a promising way, namely *layered obfuscation*. Different from mainstream obfuscation research which treats software as simple codes, we think practical obfuscation should be based on risk management

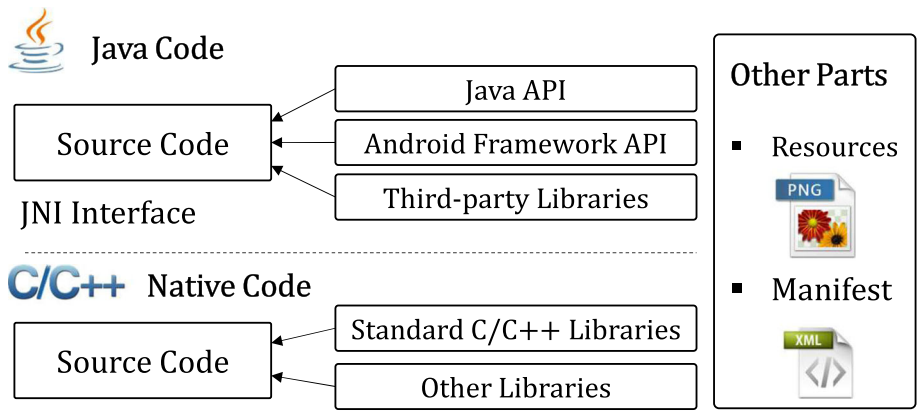


Fig. 1 General components of Android apps

and should integrate several obfuscation techniques to mitigate different risks (Gonzalez and Liñan 2019).

In practice, layered security has already been employed in protecting real-world digital assets and systems, such as digital watermarking (Barr et al. 2012) and cloud (Yildiz et al. 2009). Yet, the idea is still very preliminary for software obfuscation. Although some practical obfuscation tools (e.g., DexGuard (2018) and DexProtector (2018)) already support multiple obfuscation techniques, they do not provide a systematic way regarding how to integrate them concerning layered security.

Our contributions

In this paper, we demonstrate the concept of layered obfuscation, and we aim to help developers to adopt the idea in practice. Note that when designing layered obfuscation solutions, developers should know available obfuscation techniques as well. Such knowledge is essential for them to choose appropriate techniques and to integrate them efficiently. To meet this need, we develop a taxonomy of obfuscation techniques concerning layered security and systematically analyze the feature of each technique. In the first level of the taxonomy, we categorize obfuscation techniques into four layers based on the obfuscation targets, which are the code-element layer, software-component layer, inter-component layer, and application layer. In the second level, each layer forks into several sub-categories if the obfuscation targets can be further classified. For example, the code-element layer contains data and controls, which are two sub-categories that require different obfuscation techniques to mitigate corresponding risks. The leaf nodes of the taxonomy hierarchy are obfuscation approaches for protecting specific targets.

The rest of the paper is organized as follows. "Motivating Examples" section first demonstrates our motivation with real-world examples. "Our study approach" section introduces our approach to survey obfuscation techniques. "Code-element-layer obfuscation", "Software–component-layer obfuscation", "Inter-component-layer obfuscation", and "Application-layer obfuscation" sections survey the obfuscation techniques of different layers. "Discussion" section justifies the validity of layered obfuscation and discusses the challenges. "Related work" section discusses the related work. Finally, "Conclusion" section concludes this paper.

Motivating Examples

In this section, we discuss the obfuscation requirements of real-world software. We choose two prevalent types of software as our motivating examples, i.e., mobile apps in client-server mode and JavaScript programs in browser-server mode.

Obfuscating mobile apps

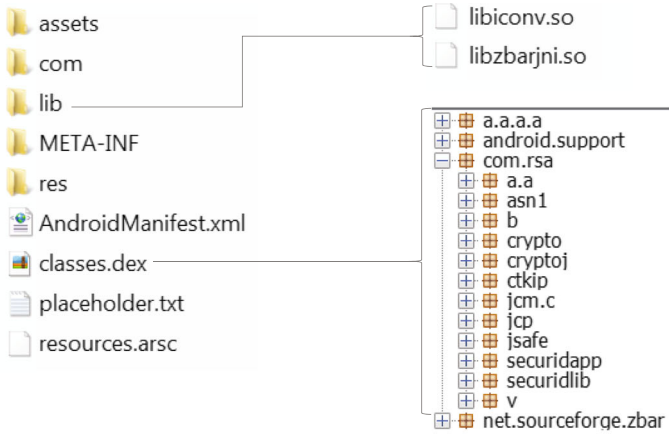
We choose *RSA SecureID Software Token* as a sample Android app to discuss the requirements of obfuscation. Figure 2a demonstrates the components of the app installation package. Its major component is `classes.dex` which contains all Java classes coded by developers. It implements user interfaces (UI) based on the APIs of Android framework and JDK. Other UI-related materials (e.g., layout, images, and texts) are mainly within the folder of `res`. Since native codes are advantageous over Java bytecodes when implementing some features, the app also employs native codes which are within the folder of `lib`. Besides, there is a manifest file and other folders to store particular data, such as licenses and fonts.

Because the main feature of the app is to generate one-time passwords, the corresponding password generation codes and seeds should be most critical for protection. However, current mainstream obfuscation techniques (e.g., lexical obfuscation and control-flow obfuscation) mainly focuses on general codes, such as Java codes or native codes. While these approaches can make the app unreadable in some sense, it is hard to evaluate the resilience of the obfuscated codes to particular reverse-engineering attacks, such as stealing the seeds. We think a promising way to tackle the problem should be based on risk management. If all the risks can be properly mitigated, developers should be confident about the obfuscation solution. Because a risk may exist in any components of the package, the obfuscation solution should integrate different techniques to mitigate corresponding risks.

Obfuscating JavaScripts

Our JavaScript example is *Performance RNN*, which is a web application that can play piano automatically based on recurrent neural networks (RNN) implemented with *tensorflow.js*. Figure 2b demonstrates the components of the web retrieved by a browser to launch the application. Similar to Android apps, these components are heterogeneous. It contains a primary HTML file (`index.html`) as the web entry, a CSS file and related pictures defining the appearance, and a JavaScript file (`bundle.js`) which implements the deep learning algorithms. Besides, there are several binary files that define an RNN model, and dozens of mp3 files to play each note of a piano.

According to the feature of the application, we infer that the key assets of the program should be the RNN model and related algorithms. Therefore, a competent obfuscation solution should at least obfuscate the model files and `bundle.js`. It may further randomize the names of the mp3 files to confuse reverse engineers. However, a better way to obfuscate the application should be based on risk analysis and risk mitigation.



(a) The components of an Android app, RSA SecureID Software Token (available on https://play.google.com/).

Type	URL
html	https://magenta.tensorflow.org/demos/performance_rnn/index.html
css	https://fonts.googleapis.com/css?family=Roboto:300,400,500,700
css	https://www.gstatic.com/external_hosted/.../mdl_css-indigo-blue-bundle.css
javascript	https://www.google.com/js/gweb/analytics/autotrack.js
png	https://storage.googleapis.com/.../magenta-logo-bottom-text2.png
javascript	https://storage.googleapis.com/.../performance_rnn/bundle.js
javascript	https://ssl.google-analytics.com/ga.js
woff2	https://fonts.gstatic.com/s/roboto/v18/KFOmCnqEu92Fr1Mu4mxK.woff2
woff2	https://fonts.gstatic.com/s/roboto/v18/KFOlCnqEu92Fr1MmWUlfBBc4.woff2
woff2	https://fonts.gstatic.com/s/roboto/v18/KFOlCnqEu92Fr1MmEU9fBBc4.woff2
mp3	https://storage.googleapis.com/.../demos/SalamanderPiano/A0v1.mp3
mp3	https://storage.googleapis.com/.../demos/SalamanderPiano/C1v1.mp3
…	
json	https://storage.googleapis.com/.../models/performance_rnn/dljs/manifest.json
binary	https://storage.googleapis.com/.../models/performance_rnn/dljs/fully_connected_biases
binary	https://storage.googleapis.com/.../models/performance_rnn/dljs/fully_connected_weights
binary	https://storage.googleapis.com/... /dljs/rnn_multi_rnn_cell_cell_0_basic_lstm_cell_bias
binary	https://storage.googleapis.com/.../dljs/rnn_multi_rnn_cell_cell_0_basic_lstm_cell_kernel

(b) The components of a Jav6aScript application, Performance RNN (available on https://js.tensorflow.org/).

Fig. 2 Motivating examples for layered obfuscation

In brief, these two examples demonstrate that practical obfuscation requirements are usually complicated. They also explain why obfuscation cannot be as secure as other security primitives. Furthermore, it indicates that layered security should be a promising way of obfuscating real-world software.

Our study approach

When designing layered obfuscation solutions for specific applications, developers should be knowledgeable about available obfuscation techniques. To meet this need, we develop a taxonomy of obfuscation techniques and survey present obfuscation techniques under the taxonomy framework. Because we aim to promote the idea of layered security in software obfuscation, the taxonomy developed in this paper is different from previous ones.

Survey Scope

This work considers all obfuscation techniques that can be adopted at the developers' side, including those obfuscation transformations for source codes, bytecodes, and assembly codes. We do not discuss other obfuscation techniques that require modifying hardware or computing systems, such as address space randomization (Bhatkar et al. 2003) and instruction set randomization (Barrantes et al. 2005).

Taxonomy of Obfuscation

Figure 3 overviews our proposed obfuscation taxonomy. In the first level of the hierarchy, we categorize present obfuscation techniques into four layers according to their obfuscation targets. The first layer is *code-element layer* which obfuscates particular elements of

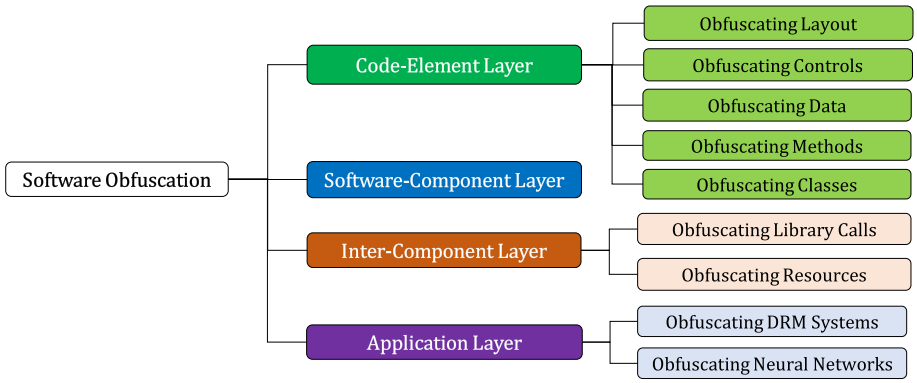


Fig. 3 The taxonomy of software obfuscation techniques for layered security

code snippets, including the layouts, controls, data, functions, and classes. The second layer is *software-component layer* which targets on an entire software component, such as a Java library or an ELF (executable file format) file. The third layer is *inter-component layer* which focuses on the interfaces (e.g., JNI) among different components of a software package. Besides, there are unique obfuscation techniques proposed for specific applications, denoted as *application layer*. A famous example of such obfuscation techniques is white-box encryption for DRM (digital right management) systems (Chow et al. 2002a). In the second level of the taxonomy, we fork each layer into several sub-categories if the obfuscation targets can be further classified in a fine-grained manner. Finally, the leaf nodes of the taxonomy hierarchy are various obfuscation strategies for particular obfuscation targets.

The underlying idea of the taxonomy is defense-in-depth, which first identifies various software assets to protect and then enumerates corresponding obfuscation techniques for protection. For example, our taxonomy can provide a quick guide to addresses different obfuscation requirements of mobile apps as discussed in "4" section. If developers are interested in protecting the lexical information, they can choose lexical obfuscation. If they need to protect the function calls between Java and native code, they should choose obfuscation techniques under the inter-component branch. Meanwhile, the design of the taxonomy hierarchy should apply to the software of different languages or forms. It is obvious that our taxonomy is also applicable to the javascript obfuscation problem discussed in "4" section.

Note that the obfuscation strategies under different branches are orthogonal to each other. Therefore, it can assist developers in locating appropriate strategies based on the characteristics of the target software. Then they can choose a combination of several obfuscation techniques by further considering their performance, such as cost, potency, and resilience. The taxonomy is different from previous work (e.g., (Collberg et al. 1997; Schrittwieser et al. 2016)) as the taxonomy is target-oriented by considering software packages composed of heterogeneous components.

Code-element-layer obfuscation

This section surveys the obfuscation techniques for specific code elements. This layer covers most of the publications in software obfuscation area. As shown in Fig. 4, according to what elements an obfuscation technique targets, we divide this category into five sub-categories: *obfuscating layouts*, *obfuscating controls*, *obfuscating data obfuscating functions*, and *obfuscating classes*.

Obfuscating layout

Layout obfuscation scrambles the layout of codes or instructions while keeping the original syntax intact. This section discusses four layout obfuscation strategies: meaningless classifiers, stripping redundant symbols, separating related codes, and junk codes.

Meaningless identifiers

This approach is also known as lexical obfuscation which transforms meaningful identifiers to meaningless ones. For most programming languages, adopting meaningful and uniform naming rules (e.g., Hungarian Notation (Simonyi 1999)) is required as a good programming practice. Although such names are specified in source codes, some would remain in the released software by default. For example, the names of global variables and functions in C/C++ are kept in binaries, and all names of Java are reserved in bytecodes. Because such meaningful names can facilitate adversarial program analysis, we should scramble them. To make the obfuscated identifiers more confusing, Chan and Yang (2004) proposed to deliberately employ the same names for objects of different types or within different domains. Such approaches have been adopted by ProGuard (2016) as a default obfuscation scheme for Java programs.

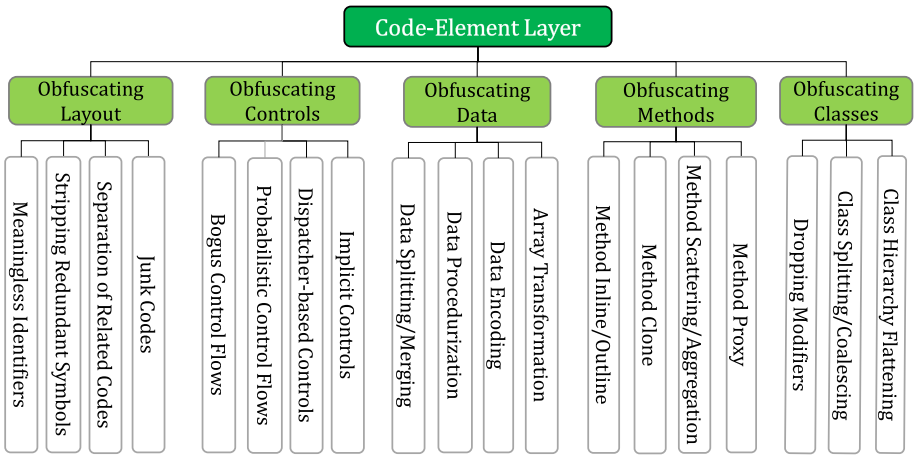


Fig. 4 The obfuscation techniques of code-element layer

Stripping redundant symbols

This strategy strips redundant symbolic information from released software, such as the debug information for most propgrams (Low 1998). Besides, there are other redundant symbols for particular formats of programs. For example, ELF files contain symbol tables which record the pairs of identifiers and addresses. When adopting default compilation options to compile C/C++ programs, such as using LLVM (Lattner and Adve 2004), the generated binaries contain such symbol tables. To remove such redundant information, developers can employ the `strip` tool of Linux. Another example with redundant information is Android smali codes. By default, the generated smali codes contain information started with `.line` and `.source`, which can be removed for obfuscation purposes (Dalla Preda and Maggi 2017).

Separating related codes

A program is more easy to read if its logically related codes are also physically close (Collberg et al. 1997). Therefore, separating related codes or instructions can increase the difficulties in reading. It is applicable to both source codes (e.g., reordering variables (Low 1998)) and assembly codes (e.g., reordering instructions (Wroblewski 2002)). In practice, employing unconditional jumps to rewrite a program is a popular approach to achieve this. For example, developers can shuffle the assembly codes and then employ `goto` to reconstruct the original control flow (You and Yim 2010). This approach is popular for assembly codes and Java bytecodes with the availability of `goto` instructions (Dalla Preda and Maggi 2017).

Junk codes

This strategy adds junk instructions which are not functional. For binaries, we can add no-operation instructions (`NOP` or `0x00`) (Dalla Preda and Maggi 2017; Marcelli et al. 2018). Besides, we can also add junk methods, such as adding defunct methods in Android smali codes (Dalla Preda and Maggi 2017). The junk codes can typically change the signatures of the codes, and therefore escape static pattern recognition.

Because layout obfuscation does not tamper with the original code syntax, it is less prone to compatibility issues or bugs. Therefore, such techniques are the most favorite ones in practice. Moreover, the techniques of meaningless identifiers and stripping redundant symbols can reduce the size of programs, which further makes them attractive (ProGuard 2016). However, the potency of the layout obfuscation is limited. It has promising resilience to deobfuscation attacks because some transformations are one-way, which cannot be reversed. However, some layout information can hardly be changed, such as the method identifiers from Java SDK. Such residual information is essential for adversaries to recover the obfuscated information. For example, Bichsel et al. (2016) tried to deobfuscated ProGuard-obfuscated apps, and they successfully recovered around 80% names.

Obfuscating controls

This type of obfuscation techniques transforms the controls of codes to increase the program complexity. It can be achieved via bogus control flows, probabilistic control flows, dispatcher-based controls, and implicit controls.

Bogus control flows

Bogus control flows refer to the control flows that are deliberately added to a program but will never be executed. It can increase the complexity of a program, e.g., in McCabe complexity (McCabe 1976) or Harrison metrics (Harrison and Magel 1981). For example, McCabe complexity (McCabe 1976) is calculated as the number of edges on a control-flow graph minus the number of nodes, and then plus two times of the connected components. To

increase the McCabe complexity, we can either introduce new edges or add both new edges and nodes to a connected component.

To guarantee the unreachability of bogus control flows, Collberg et al. (1997) suggested employing opaque predicates. They defined opaque predict as the predicate whose outcome is known during obfuscation time but is difficult to deduce by static program analysis. In general, an opaque predicate can be constantly true (P^T), constantly false (P^F), or context-dependent ($P^?$). There are three methods to create opaque predicates: numerical schemes, programming schemes, and contextual schemes.

Numerical Schemes

Numerical schemes compose opaque predicates with mathematical expressions. For example, $7x^2 - 1 \neq y^2$ is constantly true for all integers x and y. We can directly employ such opaque predicates to introduce bogus control flows. Figure 5a demonstrates an example, in which the opaque predicate guarantees that the bogus control flow (i.e., the else branch) will not be executed. However, attackers would have higher chances to detect them if we employ the same opaque predicates frequently in an obfuscated program. Arboit (2002), therefore, proposed to generate a *family* of such opaque predicates automatically, such that an obfuscator can choose a unique opaque predicate each time.

Another mathematical approach with higher security is to employ *crypto functions*, such as hash function $\mathcal{H}$ (Sharif et al. 2008), and homomorphic encryption (Zhu and Thomborson 2005). For example, we can substitute a predicate $x == c$ with $\mathcal{H}(x) == c_{hash}$ to hide the solution of x for this equation. Note that such an approach is generally employed by malware to evade dynamic program analysis. We may also employ crypto functions to encrypt equations which cannot be satisfied. However, such opaque predicates incur much overhead.

To compose opaque constants resistant to static analysis, Moser et al. (2007) suggested employing 3-SAT problems, which are NP-hard. This is possible because one can have efficient algorithms to compose such hard problems (Selman et al. 1996). For example, Tiella and Ceccato (2017) demonstrated how to compose such opaque predicates with k-clique problems.

To compose opaque constants resistant to dynamic analysis, Wang et al. (2011) proposed to compose opaque predicates with a form of *unsolved conjectures* which loop for many times. Because loops are challenging for dynamic analysis, the approach in nature should be resistant to dynamic analysis. Examples of such conjectures include Collatz conjecture, $5x + 1$ conjecture, Matthews conjecture. Figure 5b demonstrates how to employ Collatz conjecture to introduce bogus control flows. No matter how we initialize x, the program terminates with $x = 1$, and `originalCodes()` can always be executed.

Programming Schemes

Because adversarial program analysis is a major threat to opaque predicates, we can employ challenging program analysis problems to compose opaque predicates. Collberg et al. suggested two classic problems, *pointer analysis* and *concurrent programs*.

In general, pointer analysis refers to determining whether two pointers can or may point to the same address. Some pointer analysis problems can be NP-hard for static analysis or even undecidable (Landi and Ryder 1991). Another advantage is that pointer operations are very efficient during execution. Therefore, developers can compose resilient and efficient opaque predicts with well-designed pointer analysis problems, such as maintaining pointers to some objects with dynamic data structures (Collberg et al. 1998a).

Concurrent programs or parallel programs is another challenging issue. In general, a parallel region of n statements has $n!$ different ways of execution. The execution is not only determined by the program, but also by the runtime status of a host computer. Collberg et al. (1998a) proposed to employ concurrent programs to enhance the pointer-based approach by concurrently updating the pointers. Majumdar and Thomborson (2006) proposed to employ distributed parallel programs to compose opaque predicates.

Besides, some approaches compose opaque predicates with programming tricks, such as leveraging *exception*

```
int a, b;
…
if(7*a*a - 1 != b*b){
  //always true
  originalCodes();
}else{
  bogusCodes();
}
```

(a) Opaque constant.

```
int x; //for any x>0
…
while(x>1){
  if(x%2==1)
    x=x*3+1;
  else x=x/2;
  if(x==1)//always reachable
    originalCodes();
}
```

(b) Collatz conjecture.

```
int *p = &x;
int *q = &x;
if((*p)%2 == 0){
  y = x+1;
}else{
  y = x+1;
  y = y+2;
}
```

```
if((*q)%2 == 0){
  y = y+3;
  x = y+3;
}else{
  x = y+3;
}
```

(c) Dynamic opaque predicate.

Fig. 5 Control-flow obfuscation with opaque predicates

handling mechanisms. For example, Dolz and Parra (2008) proposed to use the `try-catch` mechanism to compose opaque predicates for `.Net` and Java. The exception events include division by zero, null pointer, index out of range, or even particular hardware exceptions (Chen et al. 2009). The original program semantics can be achieved via tailored exception handling schemes. However, such opaque predicates have no security basis, and they are vulnerable to advanced handmade attacks.

Contextual Schemes

Contextual schemes can be employed to compose variant opaque predicates(i.e., $\{P^?\}$). The predicates should hold some deterministic properties such that they can be employed to obfuscate programs. For example, Drape and et al. (2009) proposed to compose such opaque predicates which are invariant under a contextual constraint, e.g., the opaque predicate $x \bmod 3 == 1$ is constantly true if $x \bmod 3 : 1 \,?\, x{+}{+} : x = x{+}3$. Palsberg et al. (2000) proposed dynamic opaque predicates, which include a sequence of correlated predicates. The evaluation result of each predicate may vary in each run. However, as long as the predicates are correlated, the program behavior is deterministic. Figure 5c demonstrates an example of dynamic opaque predicates. No matter how we initialize `*p` and `*q`, the program is equivalent to $y = x + 3, x = y + 3$.

The resistance of bogus control flows mostly depends on the security of opaque predicates. An ideal security property for opaque predicates is that they require worst-case exponential time to break but only polynomial time to construct. Note that some opaque predicates are designed with such security concerns but may be implemented with flaws. For example, the 3-SAT problems proposed by Ogiso et al. (2003) are based on trivial problem settings which can be easily simplified. If such opaque predicates are implemented properly, they would be promising to be resilient.

Probabilistic control flows

Bogus control flows can make troubles to static program analysis. However, they are vulnerable to dynamic program analysis because the bogus control flows are inactive. The idea of probabilistic control flows adopts a different strategy to tackle the threat (Pawlowski et al. 2016). It introduces replications of control flows with the same semantics but different syntax. When receiving the same input several times, the program can behave differently for different execution times. The technique is also useful for combating side-channel attacks (Crane et al. 2015).

Note that the strategy of probabilistic control flows is similar to bogus control flows with contextual opaque predicates. But they are different in nature as contextual opaque predicates introduce dead paths, although they do not introduce junk codes.

Dispatcher-based controls

A dispatcher-based control determines the next blocks of codes to be executed during runtime. Such controls are essential for control-flow obfuscation because they can hide the original control flows against static program analysis.

One major dispatcher-based obfuscation approach is control-flow flattening, which transforms codes of depth into shallow ones with more complexity. Wang et al. (2000) firstly proposed the approach. Figure 6 demonstrates an example from their paper that transforms a `while` loop into another form with `switch-case`. To realize such transformation, the first step is to transform the code into an equivalent representation with `if-then-goto` statements as shown in Fig. 6; then they modify the `goto` statements with `switch-case` statements as shown in Fig. 6. In this way, the original program semantics is realized implicitly by controlling the data flow of the switch variable. Because the execution order of code blocks is determined by the variable dynamically, one cannot know the control flows without executing the program. Cappaert and Preneel (2010) formalized control-flow flattening as employing a dispatcher node (e.g., `switch`) that controls the next code block to be executed; after executing a block, control is transferred

```
int a = 1;
int b = 2;
while(a < 10){
  b = a+b;
  if(b>10){
    b--;
  }
  a++;
}
printf("%d", b);
```

(a) Source code.

```
    int a = 1;
    int b = 2;
L1: if(!(a<10))
      goto L3;
  b=a+b;
    if(!(b>10))
      goto L2;
    b--;
L2: a++;
      goto L1;
L3: printf("%d", b);
```

(b) Dismantling while.

```
int swVar = 1;
switch (swVar){
  case 1:
    a = 1; b = 2;
    swVar = 2;
    break;
  case 2:
    if(!(a<10)) swVar = 6;
    else swVar = 3;
    break;
  case 3:
    b = b+a;
    if(!(b>10)) swVar = 5;
    else swVar = 4;
    break;
  case 4:
    b--;
    swVar = 5;
    break;
  case 5:
    a++;
    swVar = 2;
    break;
  case 6:
    printf("%d", b);
    break;}
```

(c) Using switch.

Fig. 6 Control-flow flattening approach proposed by Wang et al. (2000)

back to the dispatcher node. Besides, there are several enhancements to code-flow flattening. For example, to enhance the resistance to static program analysis on the switch variable, Wang et al. (2001) proposed to introduce pointer analysis problems. To further complicate the program, Chow et al. (2001) proposed to add bogus code blocks.

László and Kiss (2009) proposed a control-flow flattening mechanism to handle specific C++ syntax, such as `try-catch`, `while-do`, `continue`. The mechanism is based on abstract syntax tree and employs a fixed pattern of layout. For each block of code to obfuscate, it constructs a `while` statement in the outer loop and a `switch-case` compound inside the loop. The `switch-case` compound implements the original program semantics, and the `switch` variable is also employed to terminate the outer loop. Cappaert and Preneel (2010) found that the mechanisms might be vulnerable to local analysis, i.e., the switch variable is immediately assigned such that adversaries can infer the next block to execute by only looking into a current block. They proposed a strengthened approach with several tricks, such as employing reference assignment (e.g., $swVar = swVar + 1$) instead of direct assignment (e.g., $swVar = 3$), replacing the assignment via `if-else` with a uniform assignment expression, and employing one-way functions in calculating the successor of a basic block.

Besides control-flow flattening, there are several other dispatcher-based obfuscation investigations (e.g., (Linn and Debray 2003; Ge et al. 2005; Zhang et al. 2010; Schrittwieser and Katzenbeisser 2011)). Linn and Debray (2003) proposed to obfuscate binaries with branch functions that guide the execution based on the stack information. Similarly, Zhang et al. (2010) proposed to employ branch functions to obfuscate object-oriented programs, which define a unified method invocation style with an object pool. To enhance the security of such mechanisms, Ge et al. (2005) proposed to hide the control information in another standalone process and employ inter-process communications. Schrittwieser and Katzenbeisser (2011) proposed to employ diversified code blocks which implement the same semantics.

Dispatcher-based obfuscation is resistant against static analysis because it hides the control-flow graph of a software program. However, it is vulnerable to dynamic program analysis or hybrid approaches. For example, Udupa et al. (2005) proposed a hybrid approach to reveal the hidden control flows with both static analysis and dynamic analysis.

Implicit controls

This strategy converts explicit control instructions to implicit ones. It can hinder reverse engineers from addressing the correct control flows. For example, we can replace the control instructions of assembly codes (e.g., `jmp` and `jne`) with a combination of `mov` and other instructions which implement the same control semantics (Balachandran and Emmanuel 2011).

Note that all existing control-flow obfuscation approaches focus on syntactic-level transformation, while the semantic-level protection has rarely been discussed. Although they may demonstrate some resilience to attacks, their obfuscation effectiveness concerning semantic protection remains unclear.

Obfuscating data

Present data obfuscation techniques focus on common data types, such as integers, strings, and arrays. We can transform data via splitting, merging, procedurization, encoding, etc.

Data splitting/merging

Data splitting distributes the information of one variable into several new variables. For example, a boolean variable can be split into two boolean variables, and performing logical operations on them can get the original value.

Data merging, on the other hand, aggregates several variables into one variable. Collberg et al. (1998b) demonstrated an example that merges two 32-bit integers into one 64-bit integer. Ertaul and Venkatesh (2005) proposed another method that packs several variables into one space with discrete logarithms.

Data procedurization

Data procedurization substitutes static data with procedure calls. Collberg et al. (1998b) proposed to substitute strings with a function which can produce all strings by specifying paticular parameter values. Drape and et al. (2004) proposed to encode numerical data with two inverse functions f and g. To assign a value v to a variable i, we assign it to an injected variable j as $j = f(v)$. To use i, we invoke $g(j)$ instead.

Data encoding

Data encoding encodes data with mathematical functions or ciphers. Ertaul and Venkatesh (2005) proposed to encode strings with Affine ciphers (e.g., Caser cipher) and employ discrete logarithms to pack words. Fukushima et al. (2008) proposed to encode the clear numbers with `exclusive or` operations and then decrypt the computation result before output. Kovacheva (2013) proposed to encrypt strings with the RC4 cipher and then decrypt them during runtime.

Array transformation

Array is one most commonly employed data structure. To obfuscate arrays, Collberg et al. (1998b) discussed several transformations, such as splitting one array into several subarrays, merging several arrays into one array,

folding an array to increase its dimension, or flattening an array to reduce the dimension. Ertaul and Venkatesh (2005) suggested transforming the array indices with composite functions. Zhu et al. (2006); Zhu (2007) proposed to employ homomorphic encryption for array transformation, including index change, folding, and flattering. For example, we can shuffle the elements of an array with $i * m \bmod n$, where i is the original index, n is the size of the original array, and m and n are relatively prime.

Obfuscating methods

Method inline/outline

A method is an independent procedure that can be called by other instructions of the program. Method inline replaces the original procedural call with the function body itself. Method outline operates in the opposite way which extracts a sequence of instructions and abstracts a method. They are good companies which can obfuscate the original abstraction of procedures (Collberg et al. 1997).

Method clone

If a method is heavily invoked, we can create replications of the method and randomly call one of them. To confuse adversarial interpretation, each version of the replication should be unique somehow, such as by adopting different obfuscation transformations (Collberg et al. 1997) or different signatures (Ertaul and Venkatesh 2004).

Method aggregation/scattering

The idea is similar to data obfuscation. We can aggregate irrelevant methods into one method or scattering a method into several methods (Collberg et al. 1997; Low 1998).

Method proxy

This approach creates proxy methods to confuse reverse engineering. For example, we can create the proxies as public static methods with randomized identifiers. There can be several distinct proxies for the same method (Dalla Preda and Maggi 2017). The approach is extremely useful when the method signatures cannot be changed (Protsenko and Muller 2013).

Obfuscating classes

Obfuscating classes shares some similar ideas with obfuscating methods, such as splitting and clone (Collberg et al. 1998b). However, since class only exists in object-oriented programming languages, such as `JAVA` and `.NET`, we discuss them as a unique category. Below we present the major strategies for obfuscating classes.

Dropping modifiers

Object-oriented programs contain modifiers (e.g., public, private) to restrict the access to classes and members of classes. Dropping modifiers removes such restrictions and make all members public (Protsenko and Muller 2013). This approach can facilitate the implementation of other class obfuscation methods.

Splitting/Coalescing class

The idea of coalescing/splitting is to obfuscate the intent of developers when design the classes (Sosonkin et al. 2003). When coalescing classes, we can transfer local variables or local instruction groups to another class (Fukushima et al. 2003).

Class hierarchy flattening

Interface is a powerful tool for object-oriented programs. Similar to method proxy, we can create proxies for classes with interfaces (Sosonkin et al. 2003). However, a more potent way is to break the original inheritance relationship among classes with interfaces. By letting each node of a subtree in the class hierarchy implementing the same interface, we can flatten the hierarchy (Foket et al. 2012).

Software-component-layer obfuscation

Now we present the obfuscation techniques which do not emphasize particular code syntax or elements. As shown in Fig. 7, such techniques include code translation, VM(virtual machine)-based obfuscation, decompilation prevention, and diversification.

Code translation

Wang et al. (2016) proposed *translingual obfuscation*, which introduces obscurity by translating the programs written in C into ProLog before compilation. Because ProLog adopts a different program paradigm and execution model from C, the generated binaries should become harder to understand. In an extreme case, Domas (2015) considered all high-level instructions should be obfuscated. He proposed *movobfuscation*, which employs only one instruction (i.e., `mov`) to compile the program. The idea is feasible because `mov` is Turing complete (Dolan 2013).

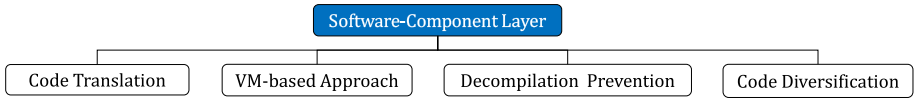


Fig. 7 The obfuscation techniques of software-component layer

VM-based Approach

VM-based obfuscation is a popular technique widely employed in practice. It converts the original machine instructions into opcode specialized for a particular virtual machine. Meanwhile, a lightweight VM is embedded in the software for runtime opcode interpretation. The original software entry is therefore replaced as a small loader that initiates the VM. There are dozens of such tools available, and the famous ones are Rewolf-x86-Virtualizer[1] (open source), VMProtect[2], Code Virtualizer[3], and Themida[4].

Since VM-based obfuscation is popular, deobfuscating such obfuscated software becomes an interest to researchers, such as (Rolles 2009; Xu et al. 2018). Hence, several methods to strengthen the protection are proposed. For example, Cheng et al. (2019) found that statistical approaches (Norouzi et al. 2016) could be effective in decoding the mapping between opcode and assembly code, and they proposed a dynamic approach that employs different mappings for different code blocks. Kuang et al. (2018) found the execution paths of state-of-the-art VM-based approaches are deterministic for the same input, which could be vulnerable. To diversify the behaviors of the obfuscated software, they propose two techniques: a randomized scheduling scheme for path selections and employing multiple VMs with different instruction sets.

Decompilation Prevention

Preventive obfuscation raises the bar for adversaries to obtain code snippets in readable formats. It is generally designed for non-scripting programming languages, such as C/C++ and Java. For such software, a decompilation or disassembly phase is required to translate machine codes (e.g., binaries) into human readable formats. Preventive obfuscation, therefore, obstructs this decoding phase by introducing decompilation errors.

Linn and Debray (2003) proposed an anti-disassembly approach for binaries. Their approach deters disassembling algorithms by inserting uncompleted instructions after unconditional jumps. In this way, the uncompleted instructions are unreachable as junk codes. If a disassembler cannot handle such uncompleted instructions, they will have troubles when separating instructions. This approach can be further strengthened with some control-flow obfuscation techniques (Popov et al. 2007). Chan and Yang (2004) proposed several lexical tricks to impede Java decompilation. The idea is to modify bytecodes directly by employing reserved keywords to name variables and functions. This is possible because only the frontend performs the validation check of identifiers. The resulting modified program can still run correctly, but it would cause troubles for decompilation tools.

Moreover, there are some encryption-based approaches which can hide the real instructions from static analysis. A typical application is the class encryption feature for Android apps (Wermke et al. 2018). By encrypting the `classes.dex`, this feature can hide the Java classes from being decompiled by popular reverse engineering tools, such as Apktool[5] and dex2jar[6].

Code Diversification

Previous obfuscation approaches focus on introducing obscurities to one software component, while code diversification generates multiple obfuscated versions of the component simultaneously (Larsen et al. 2014). Ideally, it can pose equivalent barriers for adversaries to reverse engineer each particular version. Therefore, code diversification can impede large-scale and reproductive attacks to homogeneous software (Forrest et al. 1997; Hosseinzadeh et al. 2018). It is also a technique widely employed by malware camouflage, which creates different copies of malware to evade anti-virus detection (You and Yim 2010).

Code diversification generally relies on some randomization mechanisms to introduce variance. Lin et al. (2009) proposed to generate different layout of data structures during each compilation. In this way, each compiled version contains a unique layout of data objects, such as structures, classes, and stack variables declared in functions. This can be achieved through an algorithm which automatically discovers the potential data objects that can be randomized (Xin et al. 2010). By embedding some security designs, code diversification can be resilient to specific attacks (Larsen et al. 2014; Xu et al. 2016). For example, Crane et al. (2015) proposed to randomize the tables of pointers to deter code-reuse attacks.

Inter-component-layer obfuscation

Figure 8 overviews the techniques for inter-component obfuscation. Modern software package generally contains several components, such as the components written by developers and other libraries. This phenomenon can facilitate software development and distribution, but it also raises challenging issues for obfuscation. In particular, developers cannot modify the function identifiers implemented in other libraries. However, such outsider calls provide essential information for software analysis and should be obfuscated. For example, Martín et al. (2017) showed that the function calls of third-party libraries are very effective for signature-based malware detection (Souri and Hosseini 2018).

[1] https://github.com/rwfpl/rewolf-x86-virtualizer
[2] https://vmpsoft.com/
[3] https://www.oreans.com/codevirtualizer.php
[4] https://www.oreans.com/themida.php
[5] https://ibotpeaches.github.io/Apktool/
[6] https://github.com/pxb1988/dex2jar

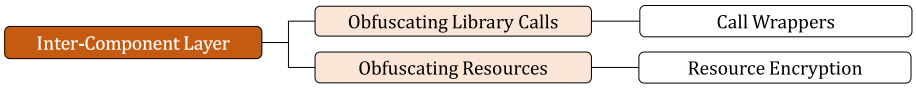


Fig. 8 The obfuscation techniques of inter-component layer

To obfuscate such information, Collberg et al. (1997) suggested substituting common patterns of function invocation with less obvious ones. However, he did not present much details. Recently, Kovacheva (2013) investigated the problem for Android apps. He proposed to obfuscate the native calls (e.g., to `libc` libraries) via a proxy, which is an obfuscated class that wraps the native functions. The feature is available in some commercial obfuscation tools, such as DexProtector (2018). Abrath et al. (2015) investigate the problem for Windows software, and they also propose to replace the original function calls via a binder. Bohannon and Holmes (2017) investigated a similar problem for Windows powershell scripts. To obfuscate an invocation command to Windows objects, they proposed to create a nonsense string first and then leverage Windows string operators to transform the string to a valid command during runtime. Besides, some state-of-the-art obfuscation tools (e.g., DexProtector (2018)) can encrypt the resource files of software packages and implement functions to decrypt them during runtime.

Application-layer obfuscation

Note that our previously discussed techniques are unrelated to the functionality of the software. In this section, we discuss several obfuscation techniques that are designed for the software with specific features. As shown in Fig. 9, we will focus on DRM systems and neural networks. However, the branch can be further extended as long as some new application obfuscation techniques are proposed.

Obfuscating DRM systems

A DRM system controls the access of users to multimedia files. The favorite solutions of DRM systems are based on content encryption. For such solutions, one critical challenge is to hide decryption keys, especially when attackers can have full access to the decryption software and the computing environment. *White-box encryption* is an obfuscation approach which can withstand key extraction attacks (Chow et al. 2002b).

In high level, a white-box encryption approach pre-evaluates all the operations related to keys and replaces corresponding codes. For example, the original DES (FIPS 46 1999) algorithm contains 16 rounds of Feistel functions. Each function `XOR`s the plaintext with a round key, and then employs a lookup table and a permutation box to produce the output. Chow et al. (2002b) proposed to substitute this procedure with a round-key-specific lookup table. In this way, it can hide both the key and round keys. To be resistant to cryptanalysis, Chow et al.proposed to further apply bijections and networked encodings for each encryption round (Chow et al. 2002b). The strategy is also applicable for AES (Chow et al. 2002a; FIPS 19 2001).

Obfuscating neural networks

Deep learning has achieved radical developments in the last decade. It is a new paradigm of programming, known as *Software 2.0*[7]. Previous studies show that the structure of neural networks is a critical factor to improve the accuracy of deep learning models. Therefore, the structural information of private machine learning models is a key intellectual property for such software. For example, our JavaScript software in "4" section contains an RNN model and should be protected.

To obfuscate deep learning models, Xu et al. (2018) proposed a simulation-based obfuscation method. The method distills the knowledge of well-trained deep learning models and reloads such knowledge into shallow networks. In this way, the shallow networks retain the same accuracy as the original models, but they have poor learning abilities. Attackers can learn very few useful settings from the simulation networks.

Discussion

Threats to validity

In this section, we justify the validity of layered obfuscation as a promising way to obfuscate real-world software. A major threat to this idea is whether there are already approaches which can obfuscate all software with security guarantee, i.e., they ensure that the essential program semantics are well protected and demonstrate adequate hardness for adversaries to recover the semantics. However, we cannot find such approaches in the literature. Below, we justify this claim from both the perspectives of practical code obfuscation and theoretical program obfuscation research.

Practical obfuscation techniques

As we have discussed in previous sections, most practical obfuscation techniques focus on obfuscating particular information. They cannot provide guarantee that the obfuscated software is secure against reverse engineering attacks.

[7] https://medium.com/@karpathy/software-2-0-a64152b37c35

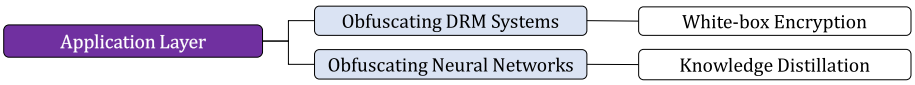


Fig. 9 The obfuscation techniques of application layer

Furthermore, real-world obfuscation practice usually adopts one obfuscation technique or combines several techniques in an ad-hoc way. For example, ProGuard (2016) is the most popular obfuscation tool for Android apps, and it is the default one embedded in Android Studio for free use. ProGuard can only obfuscate the identifiers of Java programs. Premium obfuscation tools (e.g., DexGuard (2018) and DexProtector (2018)) are more powerful, but only less than 0.16% of real-world apps employ such premium obfuscation tools (Wermke et al. 2018). From their official websites, we can find these tools support many obfuscation features, including encryption of strings, encryption of classes, hiding method calls, native code obfuscation, native code encryption, and etc. While each of these features is powerful for particular threats, there is little instruction about how to integrate them effectively. The similar situation also exists for iOS app obfuscation (Wang et al. 2018). Therefore, the taxonomy developed in this paper can provide more reference to developers regarding how to select and integrate different obfuscation techniques.

Theoretical obfuscation research

From the theoretical perspective, scientists have already found an algorithm (i.e., graded encoding) which can obfuscate all programs with a compelling security property: *indistinguishability* (Garg et al. 2013a; Zimmerman 2015; Lewi et al. 2016). Since such results may confuse readers, next, we clarify the gaps between such theoretical research and real-world obfuscation problems with a sample graded encoding mechanism.

In general, there are two phases to obfuscate a program with graded encoding: the first phase converts programs to matrix branching programs (MBP) which can be evaluated after encryption; the second phase encrypts MBPs with graded encoding mechanisms. In particular, the first phase determines the limitation of program types that can be supported by theoretical obfuscation research, and the second phase incurs large overhead.

Converting to MBP

An MBP that computes a function f is a tuple

$$MBP_f = (Input, M_{head}, (M_{i,0}, M_{i,1})_{i\in l}, M_{tail}) \tag{1}$$

Input selects a matrix $M_{i,0}$ or $M_{i,1}$ for each i according to the corresponding bit of input; M_{head} is a row vector of size w; $(M_{i,0}, M_{i,1})_{i\in l}$ are matrix pairs of size $w \times w$ that encode program semantics; and M_{tail} is a column vector of size w.

Given an input x, the MBP computes an output $MBP_f(x) \in \{0,1\}$ as follows:

$$MBP_f(x) = M_{head} \times (\prod_{i=1}^{l} M_{i,x_{input(k)}}) \times M_{tail} \tag{2}$$

Suppose the i-th matrix pair corresponds to the k-th bit of the input. If the k-th bit is 0, then $M_{i,0}$ is selected, or *vice versa*. The program output is the matrix multiplication result.

The conversion generally includes two steps: from a circuit P_f to a branching program BP_f, and from BP_f to MBP_f.

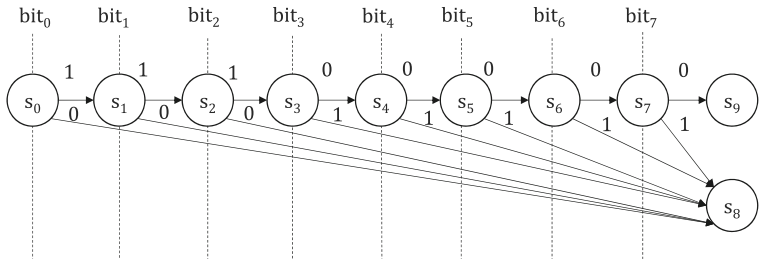
$P_f \rightarrow BP_f$: A branching program is a finite state machine. Barrington's Theorem states that we can convert any boolean formula (boolean circuit of fan-in-two, depth d) to a branching program of width 5 and length $\leq 4^d$ (Barrington 1986). For boolean formulas $P_f \in \{0,1\}$, the finite state machine has one start state, two stop states (*true* and *false*), and several intermediate states. Figure 10a demonstrates an example which converts a boolean program $i == 7$ to a branching program. Suppose i is an integer of eight bits, the boolean formula is $b_0 \wedge b_1 \wedge b_2 \wedge \neg b_3 \wedge \neg b_4 \wedge \neg b_5 \wedge \neg b_6 \wedge \neg b_7$. We need 10 states to model the branching program: eight states (s_0-s_7) that accept each bit of input, and two stop states (s_8 for *false*, and s_9 for *true*).

$BP_f \rightarrow MBP_f$: This step computes each matrix of the MBP_f. In general, M_{head} can be an all-zero row vector except the first position is 1, and M_{tail} can be an all-zero column vector except the last position is 1. $(M_{i,0}, M_{i,1})_{i\in len}$ can be constructed from the adjacency matrices of each state. Figure 10b demonstrates the matrices corresponding to the first input bit of Fig. 10a.

Following such converting approaches, the elements of resulting matrices are either 1 or 0. Kilian (1988) proposed that we can randomize these elements while retain its functionality.

$MBP_f \rightarrow RMBP_f$: We first generate $n+1$ random integer matrices RM_i and their inverse RM_i^{-1} of size $w \times w$. Then we multiply the original matrices with such random matrices as follows.

$$\begin{aligned} RM_{head} &= M_{head} \times RM_0 \\ RM_{0,0} &= RM_0^{-1} \times M_{0,0} \times RM_1 \\ RM_{0,1} &= RM_0^{-1} \times M_{0,1} \times RM_1 \\ &\cdots \\ RM_{tail} &= RM_n^{-1} \times M_{tail} \end{aligned} \tag{3}$$



(a) Branching program example.

$$\begin{bmatrix}1\\0\\0\\\vdots\\0\\0\\0\end{bmatrix}^T \quad M_0 = \begin{bmatrix}1&0&0&&0&1&0\\0&1&0&\cdots&0&0&0\\0&0&1&&0&0&0\\\vdots&&&\ddots&\vdots&&\\0&0&0&&1&0&0\\0&0&0&\cdots&0&1&0\\0&0&0&&0&0&1\end{bmatrix} \quad M_1 = \begin{bmatrix}1&1&0&&0&0&0\\0&1&0&\cdots&0&0&0\\0&0&1&&0&0&0\\\vdots&&&\ddots&&\vdots&\\0&0&0&&1&0&0\\0&0&0&\cdots&0&1&0\\0&0&0&&0&0&1\end{bmatrix} \quad \begin{bmatrix}0\\0\\0\\\vdots\\0\\0\\1\end{bmatrix}$$

head matrix — matrices for bit_0 — tail matrix

(b) Matrix branching program example.

$$Rand(M_0) = RM_{head}^{-1} \times M_0 \times RM_0 =$$

$$\begin{bmatrix}13&7&3&&3&1&0\\11&8&10&\cdots&13&5&3\\13&5&6&&12&8&3\\\vdots&&&\ddots&\vdots&&\\7&12&0&&1&13&6\\0&12&8&\cdots&0&1&15\\0&0&8&&0&0&1\end{bmatrix} \times \begin{bmatrix}1&0&0&&0&1&0\\0&1&0&\cdots&0&0&0\\0&0&1&&0&0&0\\\vdots&&&\ddots&\vdots&&\\0&0&0&&1&0&0\\0&0&0&\cdots&0&1&0\\0&0&0&&0&0&1\end{bmatrix} \times \begin{bmatrix}1&5&13&&15&3&2\\7&10&0&\cdots&2&7&6\\3&13&8&&6&7&14\\\vdots&&&\ddots&&\vdots&\\9&7&5&&8&14&5\\12&8&0&\cdots&8&13&9\\8&8&0&&0&8&1\end{bmatrix}$$

(c) Randomized matrix branching program example.

Fig. 10 The procedures to convert a program (i.e., if *x* of `int8` equals to 7) to a randomized matrix branching program

The randomization mechanism ensures that all randomization matrices RM_i would be canceled when evaluating $RMBP_f(x)$.

This phase reveals that the results of theoretical obfuscation research apply to arithmetic programs only. However, real software is more complicated which usually contains many other operations which cannot be converted to MBP directly or efficiently.

Graded Encoding

Although the randomized matrix branching program provides some security, it still suffers three kinds of attacks: partial evaluation, mixed input, and other attacks that do not respect the algebraic structure (Garg et al. 2013b). Graded encoding is proposed to defeat such attacks.

Graded encoding is based on multilinear maps. In general, a graded encoding scheme includes four components: *setup* that generates the public and private parameters of a system, *encoding* that defines how to encrypt a message with the private parameters, *operations* that declare the supported calculations with encrypted messages, and a *zero-testing function* that evaluates if the plain text of an encrypted message should be 0. *GGH scheme* is the first plausible solution to compose multilinear maps (Garg et al. 2013). It is based on ideal lattices which encodes an element e over a quotient ring $R/\mathcal{I}$ as $e+\mathcal{I}$, where $\mathcal{I} = \langle g \rangle \subset R$ is the principal ideal generated by a short vector g. The four components of GGH are defined as follows.

Setup: Suppose the multilinear level is κ. The system generates an ideal-generator g (g and g^{-1} should be short), a large enough modulus q, and denominators $\{z_i\}$ from the ring R_q. Then we publish the zero-testing parameter as $p_{zt} = [h\prod_{i=1}^{\kappa} z_i/g]_q$, where h is a small ring element.

Encoding: The encoding of an element e in set S_{z_i} is computed as : $u := [(e+\mathcal{I})/z_i]_q$.

Operations: If two encodings are in the same set (e.g., $u_1 := [c_1/z_i]_q$ and $u_2 := [c_2/z_i]_q$), then one can add them up u_1+u_2. If the two encodings are from disjoint sets, one can multiply the two encodings $u_1 \cdot u_2$.

Zero-Testing Function: A zero testing function for a level-κ encoding u is defined as

$$IsZero(u) = \begin{cases} 1 & \text{if } ||[\, u \cdot p_{zt}]_q \,||_\infty \leq q^{3/4} \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

Note that $u \cdot p_{zt} = h \cdot c/g$. If u is an encoding of 0, c should be a short vector in $\mathcal{I}$ and the product can be smaller than a threshold; otherwise, c should be a short vector in some coset of $\mathcal{I}$ and the product should be very large.

In brief, the scheme is based on noisy multilinear maps as the encoding of a value varies at different times. The only deterministic function is the zero-testing function. However, when a program becomes complex, the noise may overwhelm the signal. The size of q should be as large as possible to overwhelm the noise. This requirement largely limit the efficiency of graded encoding. Note that gradient encoding incurs polynomial overhead. Although the overhead is promising from the theoretical view, it is too large for practical usage. It has been shown that even obfuscating a 16-bit point function would result in a program of several GigBytes (Apon et al. 2014).

Other supportive work

Besides, there are investigations and obfuscation tools which coincide with our proposal of layered obfuscation. For example, Kuzurin et al. (2007) found that the security properties for obfuscating general programs might be too strong for practical scenarios. They proposed to design specific security properties for particular obfuscation scenarios, such as hiding constants or generating resilient opaque predicates. The idea is consistent with our layered obfuscation approach, i.e., an obfuscation approach cannot be secure-against-all but should only handle particular threats. Moreover, real-world obfuscation tools (e.g., Obfuscator-LLVM (Junod et al. 2015) and DexGuard (2018)) already support combinations of different obfuscation techniques, which is a characteristic of layered obfuscation solutions. However, they are still very preliminary in offering systematic combination strategies. Our paper, therefore, develops a novel taxonomy of obfuscation techniques which can assist developers in integrating them systematically.

Challenges of employing layered obfuscation

We have shown that layered obfuscation is a promising direction for achieving reliable obfuscation. Currently, there are so many obfuscation techniques available off-the-shelf, posing a great challenge to developers for choosing appropriate combinations of obfuscation techniques. Our taxonomy can provide a quick reference about available obfuscation techniques and their functionalities. But it still cannot recommend developers which techniques are their best choices. In particular, existing obfuscation papers generally employ different benchmark settings, making it difficult to compare their performance fairly. Moreover, how to design a fine-grained benchmark metric that applies to all obfuscation techniques is a challenging issue. We leave such investigations as our future work.

Related work

Our work is a pilot study of layered obfuscation. We mainly develop a taxonomy of obfuscation and survey these techniques for layered security. There are already other obfuscation surveys available, but they do not follow the layered security idea. The surveys of practical code obfuscation include (Schrittwieser et al. 2016; Drape and et al. 2009; Balakrishnan and Schulze 2005; Majumdar et al. 2006; Roundy and Miller 2012). Balakrishnan and Schulze (2005) surveyed several major obfuscation approaches for both benign codes and malicious codes. Majumdar et al. (2006) conducted a short survey that summarizes the control-flow obfuscation techniques using opaque predicates and dynamic dispatcher. Drape and et al. (2009) surveyed several obfuscation techniques via layout transformation, control-flow transformation, data transformation, language dependent transformations, etc. Roundy and Miller (2012) systematically studied obfuscation techniques for binaries, which have been frequently used by malware packers. Schrittwieser et al. (2016) surveyed the resilience of obfuscation mechanisms to reverse engineering techniques. There are also surveys of theoretical obfuscation research, including (Horváth and Buttyán 2016) and (Barak 2016). Horváth and Buttyán (2016) studied the history of cryptography obfuscation, with a focus on graded encoding mechanisms. Barak (2016) reviewed the importance of indistinguishability obfuscation. To our best knowledge, none of them follows a clear layered security approach.

Conclusion

To conclude, this work explores layered obfuscation which applies the idea of layered security to software obfuscation. To facilitate the adoption of the idea, we develop a novel obfuscation taxonomy and survey present obfuscation techniques based on the taxonomy. Our taxonomy categorizes present obfuscation techniques into four layers based on the difference of their obfuscation targets. Each layer further contains several sub-categories or obfuscation strategies. The obfuscation strategies under different branches of the taxonomy are orthogonal to each other. In this way, it can provide guidance for users when choosing obfuscation techniques for designing layered obfuscation solutions. We hope this work can inspire more investigations on layered obfuscation and encourage the development of new obfuscation techniques, which may not be secure-against-all, but can provide users more options in designing a layered obfuscation solution.

Acknowledgments
The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14210717 of the General Research Fund).

Authors' contributions
Dr. Hui Xu has accomplished most of the paper contents during his Ph.D. study, and Prof. Michael Lyu was his advisor. Dr. Yangfan Zhou and Dr. Jiang Ming are the collaborators of Dr. Hui Xu's research in software obfuscation. They gave much advice on the integrity of the taxonomy hierarchy of the paper, as well as the technical clarity of each obfuscation category. The author(s) read and approved the final manuscript.

Availability of data and materials
Not applicable.

Competing interests
Reviewers from the following institutions may have competing interests, including:

- Fudan University
- UT Arlington
- The Chinese University of Hong Kong
- Pennsylvania State University
- Peking University

Author details
[1]School of Computer Science, Fudan University, Shanghai, China. [2]School of Computer Science, Fudan University, Shanghai, China. [3]Department of Computer Science and Engineering, UT Arlington, Arlington, USA. [4]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin N. T., Hong Kong.

Received: 28 December 2019 Accepted: 25 February 2020
Published online: 03 April 2020

References

Abrath B, Coppens B, Volckaert S, De Sutter B (2015) Obfuscating windows dlls. In: 2015 IEEE/ACM 1st International Workshop on Software Protection. IEEE. pp 24–30. https://doi.org/10.1109/spro.2015.13
Apon D, Huang Y, Katz J, Malozemoff AJ (2014) Implementing cryptographic program obfuscation. IACR Cryptol ePrint Arch
Arboit G (2002) A method for watermarking java programs via opaque predicates. In: The Fifth International Conference on Electronic Commerce Research
Balachandran V, Emmanuel S (2011) Software code obfuscation by hiding control flow information in stack. In: IEEE International Workshop on Information Forensics and Security. https://doi.org/10.1109/wifs.2011.6123121
Balakrishnan A, Schulze C (2005) Code obfuscation literature survey. CS701 Constr Compilers
Barak B (2016) Hopes, fears, and software obfuscation. Commun ACM. https://doi.org/10.1145/2757276
Barak B, Goldreich O, Impagliazzo R, Rudich S, Sahai A, Vadhan S, Yang K (2001) On the (im) possibility of obfuscating programs. In: Annual International Cryptology Conference. Springer. https://doi.org/10.1007/3-540-44647-8_1
Barr JK, Bradley BA, Hannigan BT, Alattar AM, Durst R (2012) Layered security in digital watermarking. Google Patents Patent 8, US:190,901
Barrantes EG, Ackley DH, Forrest S, Stefanović D (2005) Randomized instruction set emulation. ACM Trans Inf Syst Secur. https://doi.org/10.1145/1053283.1053286
Barrington DA (1986) Bounded-width polynomial-size branching programs recognize exactly those languages in NC1. In: STOC. https://doi.org/10.1145/12130.12131
Bhatkar S, DuVarney DC, Sekar R (2003) Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: USENIX Security Symposium
Bichsel B, Raychev V, Tsankov P, Vechev M (2016) Statistical deobfuscation of android applications. In: CCS. https://doi.org/10.1145/2976749.2978422
Bohannon D, Holmes L (2017) Revoke-obfuscation: powerShell obfuscation detection using science. BlackHat
Cappaert J, Preneel B (2010) A general model for hiding control flow. In: ACM Workshop on Digital Rights Management. https://doi.org/10.1145/1866870.1866877
Chan J-T, Yang W (2004) Advanced obfuscation techniques for java bytecode. J Syst Softw. https://doi.org/10.1016/s0164-1212(02)00066-3
Chen H, Yuan L, Wu X, Zang B, Huang B, Yew P-c (2009) Control flow obfuscation with information flow tracking. In: The 42nd Annual IEEE/ACM International Symposium on Microarchitecture. https://doi.org/10.1145/1669112.1669162
Cheng X, Lin Y, Gao D, Jia C (2019) Dynopvm: Vm-based software obfuscation with dynamic opcode mapping. In: International Conference on Applied Cryptography and Network Security. Springer. https://doi.org/10.1007/978-3-030-21568-2_8
Chow S, Eisen P, Johnson H, Van Oorschot PC (2002) White-box cryptography and an AES implementation. In: International Workshop on Selected Areas in Cryptography. Springer. https://doi.org/10.1007/3-540-36492-7_17
Chow S, Eisen P, Johnson H, Van Oorschot PC (2002) A white-box DES implementation for DRM applications. In: ACM Workshop on Digital Rights Management. https://doi.org/10.1007/978-3-540-44993-5_1
Chow S, Gu Y, Johnson H, Zakharov VA (2001) An approach to the obfuscation of control-flow of sequential computer programs. In: Information Security. Springer. https://doi.org/10.1007/3-540-45439-x_10
Collberg C, Davidson J, Giacobazzi R, Gu YX, Herzberg A, Wang F-Y (2011) Toward digital asset protection. IEEE Intell. Syst. https://doi.org/10.1109/mis.2011.106
Collberg C, Thomborson C, Low D (1997) A taxonomy of obfuscating transformations, Technical report. The University of Auckland
Collberg C, Thomborson C, Low D (1998) Manufacturing cheap, resilient, and stealthy opaque constructs. In: POPL. https://doi.org/10.1145/268946.268962
Collberg C, Thomborson C, Low D (1998) Breaking abstractions and unstructuring data structures. In: IEEE International Conference on Computer Languages. https://doi.org/10.1109/iccl.1998.674154
Crane S, Homescu A, Brunthaler S, Larsen P, Franz M (2015) Thwarting cache side-channel attacks through dynamic software diversity. In: NDSS. https://doi.org/10.14722/ndss.2015.23264
Crane SJ, Volckaert S, Schuster F, Liebchen C, Larsen P, Davi L, Sadeghi A-R, Holz T, De Sutter B, Franz M (2015) It's a TRaP: table randomization and protection against function-reuse attacks. In: CCS. https://doi.org/10.1145/2810103.2813682
Dalla Preda M, Maggi F (2017) Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. J Comput Virol Hacking Tech. https://doi.org/10.1007/s11416-016-0282-2
DexGuard (2018). https://www.guardsquare.com/dexguard. Accessed Aug 2018
DexProtector (2018). https://dexprotector.com/. Accessed Aug 2018
Dolan S (2013) mov is Turing-complete
Dolz D, Parra G (2008) Using exception handling to build opaque predicates in intermediate code obfuscation techniques. J Comput Sci Technol
Domas C (2015) The movfuscator: Turning 'move' into a soul-crushing RE nightmare. REcon
Drape S, et al. (2004) Obfuscation of Abstract Data Types. Citeseer
Drape S, et al. (2009) Intellectual property protection using obfuscation. SAS
Ertaul L, Venkatesh S (2004) Jhide-a tool kit for code obfuscation. In: IASTED Conf. on Software Engineering and Applications
Ertaul L, Venkatesh S (2005) Novel obfuscation algorithms for software security. In: International Conference on Software Engineering Research and Practice. Citeseer
FIPS 19 (2001) Advanced Encryption Standard. NIST. https://doi.org/10.6028/nist.fips.197
FIPS 46 (1999) The Data Encryption Standard. NIST
Foket C, De Sutter B, Coppens B, De Bosschere K (2012) A novel obfuscation: class hierarchy flattening. In: International Symposium on Foundations and Practice of Security. Springer. https://doi.org/10.1007/978-3-642-37119-6_13
Forrest S, Somayaji A, Ackley DH (1997) Building diverse computer systems. In: The 6th IEEE Workshop on Hot Topics in Operating Systems. https://doi.org/10.1109/hotos.1997.595185
Fukushima K, Kiyomoto S, Tanaka T, Sakurai K (2008) Analysis of program obfuscation schemes with variable encoding technique. Trans Fundam Electron IEICE Commun Comput Sci. https://doi.org/10.1093/ietfec/e91-a.1.316

Fukushima K, Tabata T, Sakurai K (2003) Evaluation of obfuscation scheme focusing on calling relationships of fields and methods in methods. Commun Netw Inf Secur

Garg S, Gentry C, Halevi S (2013) Candidate multilinear maps from ideal lattices. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer. https://doi.org/10.1007/978-3-642-38348-9_1

Garg S, Gentry C, Halevi S, Raykova M, Sahai A, Waters B (2013) Candidate indistinguishability obfuscation and functional encryption for all circuits. In: Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS). IEEE. https://doi.org/10.1109/focs.2013.13

Garg S, Gentry C, Halevi S, Raykova M, Sahai A, Waters B (2013) Candidate indistinguishability obfuscation and functional encryption for all circuits (full version). In: Cryptology ePrint Archive. https://doi.org/10.1109/focs.2013.13

Ge J, Chaudhuri S, Tyagi A (2005) Control flow based obfuscation. In: ACM Workshop on Digital Rights Management. https://doi.org/10.1145/1102546.1102561

Gonzalez C, Liñan E (2019) A software engineering methodology for developing secure obfuscated software. In: Future of Information and Communication Conference. Springer. https://doi.org/10.1007/978-3-030-12385-7_72

Harrison WA, Magel KI (1981) A complexity measure based on nesting level. ACM SIGPLAN Not. https://doi.org/10.1145/947825.947829

Horváth M, Buttyán L (2016) The birth of cryptographic obfuscation-a survey

Hosseinzadeh S, Rauti S, Laurén S, Mäkelä J-M, Holvitie J, Hyrynsalmi S, Leppänen V (2018) Diversification and obfuscation techniques for software security: a systematic literature review. Inf Softw Technol. https://doi.org/10.1016/j.infsof.2018.07.007

Information Assurance Technical Framework (IATF) (2002) Release 3.1. http://www.dtic.mil/docs/citations/ADA606355. Accessed Aug 2018

Junod P, Rinaldini J, Wehrli J, Michielin J (2015) Obfuscator-LLVM: software protection for the masses. https://doi.org/10.1109/spro.2015.10

Kilian J (1988) Founding crytpography on oblivious transfer. In: STOC. https://doi.org/10.1145/62212.62215

Kovacheva A (2013) Efficient code obfuscation for android. In: International Conference on Advances in Information Technology. Springer. https://doi.org/10.1007/978-3-319-03783-7_10

Kuang K, Tang Z, Gong X, Fang D, Chen X, Wang Z (2018) Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. Comput Secur. https://doi.org/10.1016/j.cose.2018.01.008

Kuzurin N, Shokurov A, Varnovsky N, Zakharov V (2007) On the concept of software obfuscation in computer security. In: Information Security. Springer. https://doi.org/10.1007/978-3-540-75496-1_19

Landi W, Ryder BG (1991) Pointer-induced aliasing: a problem classification. In: POPL. https://doi.org/10.1145/99583.99599

Larsen P, Homescu A, Brunthaler S, Franz M (2014) Sok: Automated software diversity. In: IEEE Symposium on Security and Privacy. https://doi.org/10.1109/sp.2014.25

László T, Kiss A (2009) Obfuscating c++ programs via control flow flattening. Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica

Lattner C, Adve V (2004) Llvm: A compilation framework for lifelong program analysis & transformation. In: IEEE International Symposium on Code Generation and Optimization. https://doi.org/10.1109/cgo.2004.1281665

Lewi K, Malozemoff AJ, Apon D, Carmer B, Foltzer A, Wagner D, Archer DW, Boneh D, Katz J, Raykova M (2016) 5gen: A framework for prototyping applications using multilinear maps and matrix branching programs. In: CCS. https://doi.org/10.1145/2976749.2978314

Lin Z, Riley RD, Xu D (2009) Polymorphing software by randomizing data structure layout. In: Detection of Intrusions and Malware, and Vulnerability Assessment. Springer. https://doi.org/10.1007/978-3-642-02918-9_7

Linn C, Debray S (2003) Obfuscation of executable code to improve resistance to static disassembly. In: CCS. https://doi.org/10.1145/948109.948149

Low D (1998) Protecting java code via code obfuscation. Crossroads. https://doi.org/10.1145/332084.332092

Majumdar A, Thomborson C (2006) Manufacturing opaque predicates in distributed systems for code obfuscation. In: The 29th Australasian Computer Science Conference. Australian Computer Society

Majumdar A, Thomborson C, Drape S (2006) A survey of control-flow obfuscations. In: Information Systems Security. Springer. https://doi.org/10.1007/11961635_26

Marcelli A, Sanchez E, Squillerò G, Jamal MU, Imtiaz A, Machetti S, Mangani F, Monti P, Pola D, Salvato A, et al. (2018) Defeating hardware trojan in microprocessor cores through software obfuscation. In: the 19th Latin-American Test Symposium. IEEE. https://doi.org/10.1109/latw.2018.8349680

Martín A, Menéndez HD, Camacho D (2017) MOCDroid: multi-objective evolutionary classifier for Android malware detection. Soft Comput. https://doi.org/10.1007/s00500-016-2283-y

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng

Moser A, Kruegel C, Kirda E (2007) Limits of static analysis for malware detection. In: ACSAC. IEEE. https://doi.org/10.1109/acsac.2007.21

Norouzi M, Souri A, Samad Zamini M (2016) A data mining classification approach for behavioral malware detection. J Comput Netw Commun. https://doi.org/10.1155/2016/8069672

Ogiso T, Sakabe Y, Soshi M, Miyaji A (2003) Software obfuscation on a theoretical basis and its implementation. Trans Fundam Electron IEICE Commun Comput Sci

Palsberg J, Krishnaswamy S, Kwon M, Ma D, Shao Q, Zhang Y (2000) Experience with software watermarking. In: ACSAC. https://doi.org/10.1109/acsac.2000.898885

Pawlowski A, Contag M, Holz T (2016) Probfuscation: an obfuscation approach using probabilistic control flows. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer. https://doi.org/10.1007/978-3-319-40667-1_9

Popov IV, Debray SK, Andrews GR (2007) Binary obfuscation using signals. In: Usenix Security

ProGuard (2016). http://developer.android.com/tools/help/proguard.html. Accessed Aug 2018

Protsenko M, Muller T (2013) Pandora applies non-deterministic obfuscation randomly to android. In: The 8th International Conference on Malicious and Unwanted Software. IEEE. https://doi.org/10.1109/malware.2013.6703686

Rescorla E (2001) SSL and TLS: Designing and Building Secure Systems. Addison-Wesley Reading

Rolles R (2009) Unpacking virtualization obfuscators. In: 3rd USENIX Workshop on Offensive Technologies. WOOT

Roundy KA, Miller BP (2012) Binary-code obfuscations in prevalent packer tools. ACM Comput Surv. https://doi.org/10.1145/2522968.2522972

Schrittwieser S, Katzenbeisser S (2011) Code obfuscation against static and dynamic reverse engineering. In: Information Hiding. Springer. https://doi.org/10.1007/978-3-642-24178-9_19

Schrittwieser S, Katzenbeisser S, Kinder J, Merzdovnik G, Weippl E (2016) Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Comput Surv. https://doi.org/10.1145/2886012

Selman B, Mitchell DG, Levesque HJ (1996) Generating hard satisfiability problems. Artif Intell. https://doi.org/10.1016/0004-3702(95)00045-3

Sharif MI, Lanzi A, Giffin JT, Lee W (2008) Impeding malware analysis using conditional code obfuscation. In: NDSS

Simonyi C (1999) Hungarian notation. MSDN Libr

Sosonkin M, Naumovich G, Memon N (2003) Obfuscation of design intent in object-oriented applications. In: ACM Workshop on Digital Rights Management. https://doi.org/10.1145/947380.947399

Souri A, Hosseini R (2018) A state-of-the-art survey of malware detection approaches using data mining techniques. Human-centric Comput Inf Sci. https://doi.org/10.1186/s13673-018-0125-x

Tiella R, Ceccato M (2017) Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In: The 24th International Conference on Software Analysis, Evolution and Reengineering. IEEE. https://doi.org/10.1109/saner.2017.7884620

Udupa SK, Debray SK, Madou M (2005) Deobfuscation: reverse engineering obfuscated code. In: The 12th IEEE Working Conference on Reverse Engineering. https://doi.org/10.1109/wcre.2005.13

Wang P, Bao Q, Wang L, Wang S, Chen Z, Wei T, Wu D (2018) Software protection on the go: A large-scale empirical study on mobile app obfuscation. In: ICSE. https://doi.org/10.1145/3180155.3180169

Wang C, Davidson J, Hill J, Knight J (2001) Protection of software-based survivability mechanisms. In: DSN. https://doi.org/10.21236/ada466288

Wang C, Hill J, Knight J, Davidson J (2000) Software tamper resistance: Obstructing static analysis of programs, Technical report. University of Virginia

Wang Z, Ming J, Jia C, Gao D (2011) Linear obfuscation to combat symbolic execution. In: ESORICS. Springer. https://doi.org/10.1007/978-3-642-23822-2_12

Wang P, Wang S, Ming J, Jiang Y, Wu D (2016) Translingual obfuscation. https://doi.org/10.1109/eurosp.2016.21
Wermke D, Huaman N, Acar Y, Reaves B, Traynor P, Fahl S (2018) A large scale investigation of obfuscation use in google play. arXiv preprint arXiv:1801.02742. https://doi.org/10.1145/3274694.3274726
Wroblewski G (2002) General method of program code obfuscation, PhD thesis. Wroclaw University of Technology
Xin Z, Chen H, Han H, Mao B, Xie L (2010) Misleading malware similarities analysis by automatic data structure obfuscation. In: Information Security. Springer. https://doi.org/10.1007/978-3-642-18178-8_16
Xu D, Ming J, Fu Y, Wu D (2018) Vmhunt: A verifiable approach to partially-virtualized binary code simplification. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. https://doi.org/10.1145/3243734.3243827
Xu H, Su Y, Zhao Z, Zhou Y, Lyu MR, King I (2018) Deepobfuscation: Securing the structure of convolutional neural networks via knowledge distillation. arXiv preprint arXiv:1806.10313
Xu H, Zhou Y, Lyu MR (2016) N-version obfuscation. In: ACM International Workshop on Cyber-Physical System Security. https://doi.org/10.1145/2899015.2899026
Yildiz M, Abawajy J, Ercan T, Bernoth A (2009) A layered security approach for cloud computing infrastructure. In: International Symposium on Pervasive Systems, Algorithms, and Networks. IEEE. https://doi.org/10.1109/i-span.2009.157
You I, Yim K (2010) Malware obfuscation techniques: a brief survey. In: International Conference on Broadband, Wireless Computing, Communication and Applications. https://doi.org/10.1109/bwcca.2010.85
Zhang X, He F, Zuo W (2010) Theory and Practice of Program Obfuscation. INTECH Open Access Publisher. https://doi.org/10.5772/9632
Zhu WF (2007) Concepts and techniques in software watermarking and obfuscation, PhD thesis. ResearchSpace, Auckland
Zhu W, Thomborson C (2005) A provable scheme for homomorphic obfuscation in software security. In: The IASTED International Conference on Communication, Network and Information Security
Zhu W, Thomborson C, Wang F-Y (2006) Applications of homomorphic functions to software obfuscation. In: Intelligence and Security Informatics. Springer. https://doi.org/10.1007/11734628_18
Zimmerman J (2015) How to obfuscate programs directly. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer. https://doi.org/10.1007/978-3-662-46803-6_15

Publisher's Note