Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?

SEBASTIAN SCHRITTWIESER, St. Pölten University of Applied Sciences, Austria
STEFAN KATZENBEISSER, Technische Universität Darmstadt, Germany
JOHANNES KINDER, Royal Holloway, University of London, United Kingdom
GEORG MERZDOVNIK and EDGAR WEIPPL, SBA Research, Vienna, Austria

Software obfuscation has always been a controversially discussed research area. While theoretical results indicate that provably secure obfuscation in general is impossible, its widespread application in malware and commercial software shows that it is nevertheless popular in practice. Still, it remains largely unexplored to what extent today's software obfuscations keep up with state-of-the-art code analysis and where we stand in the arms race between software developers and code analysts. The main goal of this survey is to analyze the effectiveness of different classes of software obfuscation against the continuously improving deobfuscation techniques and off-the-shelf code analysis tools.

The answer very much depends on the goals of the analyst and the available resources. On the one hand, many forms of lightweight static analysis have difficulties with even basic obfuscation schemes, which explains the unbroken popularity of obfuscation among malware writers. On the other hand, more expensive analysis techniques, in particular when used interactively by a human analyst, can easily defeat many obfuscations. As a result, software obfuscation for the purpose of intellectual property protection remains highly challenging.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General—*Protection mechanisms*

General Terms: Security

Additional Key Words and Phrases: Software obfuscation, program analysis, reverse engineering, software protection, malware

ACM Reference Format:
Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Comput. Surv. 49, 1, Article 4 (April 2016), 37 pages.
DOI: http://dx.doi.org/10.1145/2886012

This work has been carried out partially within the scope of TARGET, the Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks. The financial support by the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development is gratefully acknowledged. The research was partially funded under Grant No. 826461 (FIT-IT) by the FFG – Austrian Research Promotion Agency.
Authors' addresses: S. Schrittwieser, Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks, Department Computer Science & Security, St. Pölten University of Applied Sciences, Matthias Corvinus-Straße 15, 3100 St. Pölten, Austria; email: sebastian.schrittwieser@fhstp.ac.at; S. Katzenbeisser, Security Engineering Group, Technische Universität Darmstadt, Hochschulstraße 10, 64289 Darmstadt, Germany; email: katzenbeisser@seceng.informatik.tu-darmstadt.de; J. Kinder, Department of Computer Science, Royal Holloway, University of London, Egham, TW20 0EX, United Kingdom; email: johannes.kinder@rhul.ac.uk; G. Merzdovnik and E. Weippl, SBA Research, Favoritenstraße 16, 1040 Vienna, Austria; emails: {gmerzdovnik, eweippl}@sba-research.org.

DOI: http://dx.doi.org/10.1145/2886012

1. INTRODUCTION

The development of code obfuscation techniques is driven by the desire to hide the specific implementation of a program from automatic or human-assisted analysis of executable code. Its roots date back to the early days of programming when the main purpose of obfuscation was to place hidden functionality into programs—often just for hiding a secret message to surprise users who reveal it by coincidence (e.g., Easter eggs). Particularly creative ways to obscure functionality were even rewarded in competitions like the International Obfuscated C Contest, which has been held since 1984.[1]

By the late 1980s, malware became a major reason for the steady refinement of code obfuscation techniques. For instance, in 1986 the Brain computer virus, which is believed to be the first computer virus for MS-DOS, obfuscated the functionality of its code by intercepting reads of the virus binary to return innocuous code [Avoine et al. 2007]. These early occurrences of software obfuscation marked the beginning of an arms race between developers and analysts. Since then, software protection researchers have been developing more and more sophisticated obfuscation techniques to hide behavior of code, while analysts have been using increasingly complex code analysis techniques to defeat obfuscations.

The underground economy and its demand for stealthy malware is still one of the major driving forces behind the development of obfuscation techniques. The other leading application area of code obfuscation is the protection of commercial software against reverse engineering [Grover 1992]. The motivation for reverse engineering can be diverse: An analyst might be interested in extracting some secret information from the program code that should not be revealed to the user. This secret might be a cryptographic key, a sophisticated algorithm considered a trade secret, or credentials for a remote service. Typical examples include media players that store secret keys or incorporate proprietary algorithms like the content scramble system (CSS) cipher to play back copy-protected content. Another motivation for reverse engineering is the modification of software to change its behavior. An analyst might want to use hidden program functionality, uncover firmware features that are disabled in low-cost devices, or interface commercial software with own products; all these actions will likely interfere with the business model of the software vendor.

Techniques originally developed for malware have experienced a comeback in the field of attack prevention. Since the early 1990s polymorphic engines had been widely used in malware development to generate different-looking versions of malicious code to evade signature-based detection [Nachenberg 1997; Song et al. 2010]. Software diversity is a related concept for delivering syntactically different but semantically identical versions of applications to different users [Franz 2010; Davi et al. 2012]. This way, automated attacks developed against one instance of a program are thus less likely to work against a different obfuscated version [Forrest et al. 1997; Anckaert et al. 2004, 2006].

On the analyst's side, research in the area of code analysis progressed significantly over the past several years. Disassemblers, which extract the executable assembly code from binaries, became increasingly sophisticated, implementing complex heuristics or expensive static analysis methods to reconstruct code from a potentially obfuscated binary as precisely as possible. In addition, research tools such as JAKSTAB [Kinder and Veith 2008] or BAP [Brumley et al. 2011] directly analyze binary code to precisely reconstruct control flow and reason about the behavior of the code. Systems for dynamic malware analysis, such as Anubis [Bayer et al. 2006], allow a detailed analysis of the runtime behavior of a piece of code.

[1] http://www.ioccc.org/1984/ (accessed February 25, 2015).

Since the groundbreaking work of Collberg et al. [1997] on a taxonomy of obfuscation techniques, a vast number of code obfuscation schemes have been proposed in the literature. Nevertheless, the security and effectiveness of these techniques remain controversially discussed topics. Indeed, early theoretical results for obfuscation that is secure in a cryptographic sense were negative. Barak et al. [2001] showed that it is impossible to construct a universal obfuscator that is applicable to any program; positive results are known for very restricted classes of functionality such as point functions [Lynn et al. 2004; Wee 2005; Canetti and Dakdouk 2008]. Other works on theoretical aspects of code obfuscation include those of Dalla Preda and Giacobazzi [2005], Goldwasser and Rothblum [2007], and Giacobazzi [2008]. While addressing the theoretical understanding of obfuscation, these results have little to say about the practical effectiveness of obfuscation. Recently, Garg et al. [2013] presented a first candidate for general-purpose obfuscation (*indistinguishability obfuscation*). While it is unclear how useful this progress is to hide secrets in a program in practical settings, it still brought new momentum to the discussion and encouraged follow-up work [Brakerski and Rothblum 2014; Barak et al. 2014; Bitansky et al. 2014].

Code obfuscation is widely employed in practice: For example, almost all newly discovered malware comes with some form of obfuscation to hide its functionality, and commercial software products such as Skype or Digital Rights Management (DRM) engines use obfuscation techniques as part of their protection portfolio. Consequently, a plethora of obfuscation techniques have been described in the literature (e.g., Collberg et al. [1997], Wang et al. [2000], Dedić et al. [2007], and Jakubowski et al. [2007]); most of them try to "raise the bar" for reverse engineering attempts but do not come with a rigorous security analysis, let alone a proof. Some attempts have been made to quantify the additional complexity that is added to an executable by employing code obfuscation techniques (for example, see Madou et al. [2006]); however, it remains unclear whether such notions indeed capture all security properties of obfuscation correctly. While this is clearly unsatisfactory from a theoretic point of view, it is probably the best one can achieve with current knowledge.

Still, it remains largely unexplored to what extent today's code obfuscation techniques can keep up with the progress in code analysis and where we stand in the arms race between developers and analysts. The main goal of this survey is thus to provide a comprehensive picture of the state of the art in code obfuscation and of its effectiveness against deobfuscation techniques and code analysis tools that are available today.

To this end, we first build a classification of analysis scenarios in Section 2; in particular, we pair each analysis method with a specific analysis goal to define a set of *scenarios* that form our basic model of the code analyst. Subsequently, we describe and categorize existing code obfuscation techniques in Section 3. There is a large body of literature on obfuscation and analysis, and we do not claim to survey it in all its breadth; instead, we try to paint a representative picture of the state of the art in both fields.

In all of our analysis scenarios, we assume that the analyst receives binary or byte code. Thus, we limit ourselves to obfuscations that *affect* the binary or byte code representation of a program. We make no distinction whether the obfuscation is *applied* before, during, or after compilation, however. We decided to leave obfuscation techniques targeting the readability of source code (such as the removal of comments or renaming of variables) out of scope, even though we acknowledge the importance of such obfuscations for languages such as JavaScript.

Section 4 discusses the state of the art in program analysis methods and particularly focuses on their capabilities and limits. Section 5 uses the described classifications to assess the security of code obfuscation techniques against the different scenarios, taking into account recently published attacks as well as recent off-the-shelf program analysis tools. We rely on published results where possible and apply our own judgment

where we could not find literature addressing a particular scenario. Finally, Section 6 concludes the article.

2. ANALYSIS SCENARIOS

In this section, we detail our methodology and introduce the analysis scenarios we use throughout the rest of the article to assess the strength of obfuscation techniques against specific adversaries with specific goals. This allows us to characterize the protection a developer may want to achieve at a finer granularity than the blackbox notion typically used in the formal analysis of software obfuscation [Barak et al. 2001].

2.1. Methodology of This Survey

We classify the feasibility of real-life analysis scenarios on programs in the presence of code obfuscation, based on a careful analysis of the literature on program analysis and reverse engineering. For our classification, we use a number of analysis scenarios consisting of an analysis technique and a goal that an analyst wants to achieve.

We assume that a developer implements the strongest possible obfuscator of one particular class of obfuscations. We do not consider second order effects, that is, combinations of different types of obfuscations. A systematic analysis of how obfuscation techniques compose with one another would be challenging already just due to the sheer number of possible combinations. Still, it certainly makes an interesting and relevant topic for future work, since many commercial obfuscators employ (and indeed recommend to use) multiple obfuscations at the same time.

We make a similar assumption on the side of the analyst and evaluate the strength of obfuscations against four code analysis categories (introduced in Section 2.2) individually. Approaches that involve the staged application of multiple techniques are represented by the category of their first analysis stage; for instance, pattern matching on disassembled code requires an at least partially successful disassembly by static analysis. Hybrid techniques that unite aspects of multiple categories are sorted into the category they share the most properties with. Our strongest category involves a human analyst who may use multiple techniques of different types; our model here assumes that the analyst will apply techniques from either category and switch to a different one if they do not make any more progress towards the analysis goal.

As stated above, our evaluation of the effectiveness of different classes of software obfuscation schemes against the defined analysis scenarios is based on a literature review. For each combination of an analysis scenario and a specific class of obfuscations (see Section 3), we sought out literature specifically addressing that combination. Sometimes we could directly report published results of analyzing particular obfuscations, sometimes we had to make our own inferences from the available information. Our results are not derived from a formal analysis and are necessarily open to interpretation. Nevertheless, we believe that our scenarios capture the relevant analysis context and that our classification presents an accurate snapshot of the current state in the arms race between code obfuscation and analysis.

2.2. Code Analysis Categories

We categorize code analysis techniques in four general classes. An analyst can use different analyses depending on their goal and their available resources and time. For example, a human reverse engineer who tries to understand a piece of code of a competitor may afford spending time and effort on highly complex and time-consuming analyses; in contrast, an antivirus vendor, who has to timely analyze hundreds of thousands of different malware samples each day, may be required to resort to very lightweight and thus limited analysis techniques that at the same time ensure a low false-positive rate.

Pattern matching. Pattern matching is the simplest and fastest form of code analysis; it is a syntactic analysis performed on the program binary or byte code. Techniques in this category include identification of static sequences of instructions, regular expressions, or machine learning-based classifiers for binary data (see Section 4.1).

Automated static analysis. Static analysis inspects a program (on binary or byte code level) without actually executing it. In contrast to syntactic pattern matching, static analysis reasons about the program semantics. Simple forms of static analysis include disassemblers that interpret branch targets. Static analysis is frequently used to reconstruct high-level information about programs, such as their control flow graph (see Section 4.2).

Automated dynamic analysis. Dynamic analysis runs a program to observe its actions and/or collect its flow of information. Dynamic analysis has the advantage of being able to very precisely reason about the program behavior along the observed traces. However, the data gathered from one or several runs of a program does not necessarily allow to draw conclusions about the behavior of the entire program (see Section 4.3).

Human-assisted analysis. As the most capable "analysis," we consider a human analyst who performs a tool-assisted exploration of a piece of code; the tools can be based on all of the other analysis types. Typically, this process is referred to as reverse engineering, where the analyst aims at understanding the program structure and behavior with the help of a variety of tools (see Section 4.4).

2.3. Analyst's Aims

We now systematically categorize and characterize the possible motivations of an analyst for analyzing software, in order of increasing complexity. We believe that almost all analyst goals observed in practice fit in one of these four general categories. As a running example to illustrate the goals throughout the section, we use a program implementing a cryptographic algorithm with an embedded secret key. Furthermore, we present other examples for each category to demonstrate the practicability of this classification.

Finding the location of data. The analyzer wishes to retrieve some data embedded in the program in its original, nonobfuscated representation from the obfuscated program. In our running example, the analyzer may want to extract the secret cryptographic key from the obfuscated program to be able to decrypt data in a different context than provided by the application (e.g., to circumvent DRM policies). Other typical examples that fall into this category are the extraction of licensing keys, certificates, credentials for remote services, and device configuration data.

Finding the location of program functionality. The analyzer aims at identifying the entry point of a particular function within an obfuscated program. In our running example, the analyzer may want to find the entry point of a cryptographic algorithm in the obfuscated program to subsequently analyze it. Another aim could be to find the exact location of a copy protection mechanism (such as a check for the presence of a hardware dongle or the validation of a licensing key) in order to circumvent it. Furthermore, finding the code representation of a particular functionality of a program can be useful for manual reverse engineering on small areas of the program. More generally, one may ask the question whether a program implements a particular functionality at all, such as the AES (Advanced Encryption Standard) algorithm, or simply whether a program is malicious or not.

Another related aim of the analyst might be to modify the behavior of a program in a particular way (e.g., bypassing a copy protection mechanism). However, we consider

this to be out of scope of this work, because it falls into the field of tamper-proofing rather than obfuscation (e.g., Horne et al. [2002] and Chang and Atallah [2002]). Still, *finding the location of program functionality* is a fundamental prerequisite of this aim.

Extraction of code fragments. The analyzer aims at extracting a piece of code including all possible dependencies that implement a particular functionality from the obfuscated program. In our running example, the analyzer's aim can be the extraction of the cryptographic algorithm in order to build a custom decryption routine. Note that for this purpose it is not necessary to fully understand the code; just using it in a new application may be enough. This approach has been used for breaking DRM implementations. Instead of understanding how exactly the decryption routine embedded into a player works, it is simply extracted and included in a counterfeit player, which decrypts the digital media without enforcing the contained usage policies. Another aim of the analyst may be the extraction of fragments of commercial software of competitors. In some analysis scenarios code does not need to be extracted, but a functional component of an executable is accessed at runtime (*in-situ reuse*) [Miles et al. 2012]. This technique is comparable to the concept of return oriented programming, which reuses code fragments to create new functionality [Shacham 2007].

Understanding the program. The analyzer wishes to fully understand a code fragment or even the entire obfuscated program. This requires that the analyst must be able to remove the applied obfuscation techniques and gain full understanding of the original, nonobfuscated program or a nontrivial fragment of it. In our running example, the analyzer may want to understand how a proprietary cipher embedded into the obfuscated program works in order to start cryptanalysis attempts. Other motivations for trying to understand a program can be the desire of an analyst to find vulnerabilities, to correct flaws in software for which the source code is not available, and create new programs that are compatible with proprietary software. Finally, a major driving force for human-assisted reverse engineering is to gain understanding of proprietary implementation details such as file formats or protocols, which often constitutes intellectual property theft.

2.4. Scenarios

By combining each code analysis technique with each of the analyst's aims, we arrive at the analysis scenarios that form the basis of our survey.

Not all combinations are reasonable according to our definitions. While pure pattern matching can clearly help to locate a code fragment, its extraction will then either (a) require additional static analysis for dependency analysis and thus fall into another category or (b) be trivial and thus identical in difficulty. Similarly, pattern matching does not lend itself to code understanding, so we leave out both combinations and consider a total of 14 scenarios. An overview of the literature that describes code analysis techniques based on these scenarios is provided in Table I. In the following, we explain all 14 scenarios in more detail.

Locating data through pattern matching. A matching algorithm is used to determine the existence and location of data that conforms to a pattern specification. Patterns for the automated identification of data inside a program describe the structure of the data such as its length or data type, its environment in the form of the code surrounding it, or even properties such as its entropy.

Locating code through pattern matching. Particular code fragments are detected by pattern matching, for example, on instruction sequences with or without wildcards. When pattern matching is applied to the outcome of a static or dynamic analysis

Table I. Literature on Code Analysis Categorized Based on the 14 Scenarios

	Pattern Matching	Automatic Static	Automatic Dynamic	Human Analysis
Locating Data	[Moser et al. 2007b]	[Shamir and Van Someren 1999] [Kinder and Veith 2008] [Kinder 2012]	[Cozzie et al. 2008] [Lin et al. 2010] [Slowinska et al. 2011] [Zhao et al. 2011]	[Jacob et al. 2003] [Link et al. 2004] [Billet et al. 2005] [Wyseur and Preneel 2005] [Goubin et al. 2007] [Piazzalunga et al. 2007] [Wyseur et al. 2007] [Michiels et al. 2009] [Saxena et al. 2009] [Wyseur 2009] [De Mulder et al. 2010]
Locating Code	[Newsome et al. 2005] [Moser et al. 2007b] [Tang and Chen 2007] [Griffin et al. 2009] [Dalla Preda et al. 2011]	[Flake 2004] [Harris and Miller 2005] [Bruschi et al. 2006a] [Bruschi et al. 2006b] [Chouchane and Lakhotia 2006] [Dalla Preda et al. 2006] [Walenstein et al. 2006] [Bilar 2007] [Karnik et al. 2007] [Nagarajan et al. 2007] [Popov et al. 2007] [Gao et al. 2008] [Coogan et al. 2009] [Treadwell and Zhou 2009] [Tsai et al. 2009] [Jacob et al. 2012] [Kinder 2012] [Bourquin et al. 2013]	[Deprez and Lakhotia 2000] [Madou et al. 2005] [Zhang and Gupta 2005] [Royal et al. 2006] [Wilde and Scully 2006] [Moser et al. 2007a] [Brumley et al. 2008] [Sharif et al. 2008] [Song et al. 2008] [Guizani et al. 2009] [Li et al. 2009] [Sharif et al. 2009] [Webster and Malcolm 2009] [Comparetti et al. 2010] [Debray and Patel 2010] [Yin and Song 2010] [Coogan et al. 2011] [Gröbert et al. 2011] [Calvet et al. 2012] [Zeng et al. 2013]	[Madou et al. 2006b] [Madou et al. 2006c] [Rolles 2009] [Quist and Liebrock 2009]
Extracting Code	*invalid scenario*	[Sneed 2000] [Christodorescu et al. 2007]	[Christodorescu et al. 2007] [Leder et al. 2009] [Sharif et al. 2009] [Caballero et al. 2010] [Kolbitsch et al. 2010] [Zeng et al. 2013]	[Ning et al. 1993] [Canfora et al. 1994] [Field et al. 1995] [Cimitile et al. 1996] [Lanubile and Visaggio 1997] [Canfora et al. 1998] [Danicic et al. 2004] [Fox et al. 2004] [Danicic et al. 2005]
Understanding Code	*invalid scenario*	[Rugaber et al. 1995] [Majumdar et al. 2006] [Raber and Laspe 2007] [Guillot and Gazet 2010]	[Udupa et al. 2005]	[Biondi and Desclaux 2006] [Eagle 2008] [Myska 2009] [Kholia and Wegrzyn 2013]

(e.g., on disassembled or normalized code [Christodorescu et al. 2007]), we classify the analysis as static or dynamic, respectively.

Locating data through static analysis. A static analysis interprets the semantics of a program to locate particular data. For example, the reconstruction of the possible arguments of function calls can reveal a cryptographic key that is used to decrypt DRM-protected media or sent to an external device, or it could determine a token that is transmitted over the network.

Locating code through static analysis. A static analysis that determines code that could be executed at runtime. Examples are disassemblers that interpret control flow

instructions using heuristics or formal methods. The control flow graph often allows us to recognize particular structures; for example, the control flow characteristics of a cryptographic algorithm can reveal its location in a binary.

Extracting code through static analysis. A static analysis technique (such as def-use chains) analyzes the dependencies of a piece of code to compute a self-contained slice that can be extracted and run on its own. Most static analysis techniques will result in extracting a superset of the required code (see Section 4.2).

Understanding code through static analysis. This scenario captures static deobfuscation techniques that are able to transform fragments or all of the obfuscated code into a low- or high-level representation from which a human analyst can understand the program functionality with reasonable effort.

Locating data through dynamic analysis. A dynamic analysis observes the data accessed or stored by a program at runtime and uses these observations to locate particular data, for example, in the parameters of system calls or particular memory locations.

Locating code through dynamic analysis. Dynamic analysis reveals the behavior of a program and its interaction with the environment (e.g., system calls) at runtime. In this scenario, a particular functionality is located through its specific runtime behavior.

Extracting code through dynamic analysis. A dynamic analysis locates a piece of code and also determines its dependencies along the observed traces. Because the code extracted from one run may not contain everything required for another run of the program, the resulting control flow graph is a subset of all possible execution paths (see Section 4.3).

Understanding code through dynamic analysis. Analogously to the scenario of understanding code through static analysis, this scenario targets automated dynamic deobfuscation techniques that are able to transform the obfuscated code into a representation from which a human analyst can understand the program's functionality with reasonable efforts.

Human-assisted analysis (four scenarios). Static and/or dynamic approaches assisted by a human analyst aim at getting full understanding of a particular aspect of the program. This aspect can be data in its pure form (data deobfuscation), the location of code implementing a particular functionality or its dependencies, as well as the entire program itself.

3. SOFTWARE OBFUSCATION

In this section, we briefly describe various code obfuscation schemes from the literature and classify them into the three categories of *data obfuscation*, *static code rewriting*, and *dynamic code rewriting*. Many of the described obfuscation techniques appeared first in malware samples. Consequently, it is hard to pay tribute to the original source; we did this wherever possible. More details on early techniques are given by Collberg et al. [1997] and Collberg and Nagra [2009].

3.1. Data Obfuscation

Code obfuscation techniques of this category modify the form in which data are stored in a program to hide it from direct analysis. Usually, data obfuscation also requires the program code to be modified, so the original data representation can be reconstructed at runtime. Many data obfuscation techniques were first described by Collberg et al. [1998a].

Reordering of data. Variables can be split into two or more pieces in order to make it more difficult for an analyst to identify them. The mapping between an actual value of a variable and its split representation is managed by two functions. One is executed at obfuscation time, and the other one reconstructs the original value of a variable from its split parts at runtime. For example, Boolean variables can be obfuscated by splitting them into multiple Boolean values. At runtime, the variable's actual value is retrieved by performing a specific Boolean operation (such as a logical XOR) over the parts of the variable. Other data types such as integers and string variables can be obfuscated in a similar way. In contrast to variable splitting, variable merging combines two or more variables into one.

To obfuscate the structure of an array, it can be *split* into two or more subarrays. Conversely, multiple arrays can be *merged* into one. *Folding* (increasing the number of dimensions of the array) and *flattening* (decreasing the number of dimensions) are similar techniques which can be used for obfuscating data stored in arrays.

Obfuscating the structure of data by reordering its components to decrease locality (logically related items are physically close in the binary) is another fundamental obfuscation technique [Collberg et al. 1998a]. For example, this obfuscation is often applied to cryptographic keys stored within commercial software.

A low-level implementation of data reordering for obfuscation was introduced by Anckaert et al. [2009]. By redirecting memory accesses though a software-based dispatcher, the order of data in memory can be shuffled periodically, making its identification and analysis more difficult.

Encoding. Static data (such as strings) within binaries contains useful information for an analyst. Under this obfuscation technique, data are converted to a different representation with some special encoding function to mitigate the need of storing the static data in cleartext in the binary. At runtime, the inverse function is used to decode the data. A variant of data encoding that works via mixed-mode computation over Boolean-arithmetic algebras was introduced by Zhou et al. [2007].

Converting static data to procedures. This obfuscation method replaces static data with a function that calculates the data at runtime. For example, a string object can be built at runtime, so an analyst is not able to extract its value by examining the binary.

An extreme form of this obfuscation method is whitebox cryptography. Its basic idea is to merge a secret key with elements of the cipher (e.g., the S-boxes), so the key cannot be found in the binary anymore. The first whitebox implementations of Data Encryption Standard (DES) [Chow et al. 2003a] and AES [Chow et al. 2003a] were proposed by Chow et al., followed by other approaches [Link and Neumann 2005; Wyseur and Preneel 2005; Bringer et al. 2006]. However, all published whitebox algorithms have been broken so far (for more details, please refer to Section 5).

3.2. Static Code Rewriting

A static rewriter is similar to an optimizing compiler, as it modifies program code during obfuscation but allows its output to be executed without further runtime modifications. Strictly speaking, all data obfuscation techniques described above would also fall into the category of static code rewriting. However, as the obfuscation targets are distinct (data vs. binary code) and require specific obfuscation techniques, we use separate categories for data obfuscation and static code rewriting. In malware obfuscation the term *metamorphism* describes automated mutation of binary code through static code rewriting techniques applied to its disassembled representation.

Replacing instructions. Any behavior of a program can be implemented in multiple ways, and instructions or sequences of instructions can be replaced with syntactically

different, yet semantically equivalent, code. For example, on the Intel x86 platform the instructions `MOV EAX, 0` and `XOR EAX, EAX` are equivalent and can be replaced with each other. More complex obfuscations of this type include the replacement of CALLs with a combination of `PUSH` and `RET` instructions [Lakhotia et al. 2010]. De Sutter et al. [2009] replaced infrequently used opcodes with blocks of more frequently used ones to reduce the total number of different opcodes used in the code and to normalize their frequency. Similarly, shellcode for application exploits can be transformed into innocuous-looking sequences of input characters by tools such as SCMorphism.[2] Mason et al. [2009] demonstrated how shellcode can even be encoded into a representation that looks similar to English prose. Related concepts are used to hide network traffic by transforming an encrypted payload into a format that resembles common network protocols such as HTTP [Dyer et al. 2013].

Potentially malicious code can also be hidden in side effects of innocent-looking sequences of instructions [Schrittwieser et al. 2013]. A side effect can be any effect on the state of the underlying machine that is not covered by the analysis model (e.g., the state of the flags register). Similar concepts have also been discussed in the context of shellcode obfuscation.[3]

Opaque predicates. A predicate (Boolean-valued function) is opaque if its outcome is known to the obfuscator at obfuscation time but difficult to determine for a deobfuscator [Collberg et al. 1997, 1998b]. Opaque predicates are used to make static reverse engineering more complex by introducing an analysis problem which is difficult to solve without running the program. The prime example for the use of opaque predicates is the obfuscation of a program's control flow graph by adding conditional jumps that are dependent on the result of opaque predicates.

To prevent an analyst from identifying opaque predicates through their static behavior across a large number of program executions, refined concepts for opaque predicates were developed. Palsberg et al. [2000] introduced the concept of *dynamic opaque predicates*. It uses a set of correlated opaque predicates that all evaluate to the same result in one run, but they may all evaluate to the same different result in other runs of the program. Majumdar and Thomborson [2006] described *temporally unstable opaque predicates* for which evaluations at multiple points inside the program lead to different results.

Inserting dead code. The term "dead code" refers to code blocks which are not or simply cannot be reached in the control flow graph and thus never get executed [Collberg et al. 1997]. Inclusion of such code can make the analysis of a program more time consuming as it increases the amount of code that has to be analyzed. For making the identification of dead code more difficult, opaque predicates that always resolve to either `true` or `false` can be used.

Inserting irrelevant code. Cohen [1993] described the concept of irrelevant ("garbage") code. Sequences of instructions that do not have an effect on the execution of a program can be inserted into the code to make analysis more complex. The most simple form of irrelevant code are `NOP` instructions which do not modify the program's state. In contrast to dead code, irrelevant code can be reached by the control flow of the program and is executed at runtime, however, without any effect on the program state.

Reordering. Similarly to data structures, also expressions and statements can be reordered to decrease locality, whenever the order does not affect the program behavior.

[2]http://www.kernelhacking.com/rodrigo/scmorphism/HowItWorks.txt (accessed June 07, 2015).
[3]http://www.securityfocus.com/archive/82/327100/2009-02-24/1 (accessed June 07, 2015).

Such techniques were originally introduced for code optimization [Bacon et al. 1994] but also apply in the context of obfuscation.

This concept can be taken even further to move parts of the code or functionality into different modules or programs, as was done in the Stuxnet malware [Matrosov et al. 2010].

Loop transformations. Many loop transformations have been designed to improve the performance and space usage of loops [Bacon et al. 1994]. Some of them increase the complexity of the code and can therefore be used for obfuscation purposes. *Loop tiling* was originally designed to optimize the cache behavior of code. It breaks up the iteration space of a loop and creates inner loops that fit in the cache. In *loop unrolling*, originally developed to improve performance, the body of the loop is replicated one or more times to reduce the number of loop iterations. The *loop fission* method splits a loop into two or more loops with the same iteration space and spreads the loop body over these new loops.

Function splitting/recombination. Function cloning describes the concept of splitting the control flow in two or more different paths that look different to the analyst, while they are in fact semantically equivalent. Another transformation type merges the bodies of two or more (similar) functions. The new method has a mixed parameter list of the merged functions and an extra parameter that selects the function body to be executed.

The related idea of overlapping functions—where the binary code of one function ends with bytes that also define the beginning of another function—is commonly used by compilers for optimization purposes and can also by used to confuse a disassembler. A similar but more sophisticated concept was introduced by Jacob et al. [2007]: two independent code blocks are interweaved in a way that, depending on the entry and exit points of the merged code, different functionality is executed.

Aliasing. Inserting spurious aliases (i.e., pointers to memory locations) can make code analysis more complex, as the number of possible ways for modifying a particular location in memory increases [Horwitz 1997; Ramalingam 1994]. These pointer-references can also be used as indirections to complicate the reconstruction of the control flow graph of a program in static analysis scenarios [Wang et al. 2001].

Name scrambling. Modifying identifier names such as the ones of variables and methods and replacing them with random strings is a prime example for source code obfuscation, which is not covered in our work. While binary code usually does not contain identifier names any more, byte code preserves some of the identifier names. For example, Java byte code contains class, field, and method names. By substituting expressive names with random strings, semantic information that can be important for a human analyst is removed.

Control flow obfuscation. This class of obfuscations aims at obfuscating the program control flow graph. The control flow flattening obfuscation completely obscures the links between basic blocks. Wang et al. [2000] described *chenxification*, which puts the basic blocks of a program into a large switch-statement (called dispatcher) that decides where to jump next based on an opaque variable. Control flow flattening using a central dispatcher was also described by Chow et al. [2001]. A similar concept by Linn and Debray [2003] uses a so-called *branch function* to obfuscate the targets of CALL instructions. All calls are forced to pass through the branch function, which directs the control flow to the actual target based on a call table. Popov et al. [2007] proposed to replace control transfer instructions by traps that cause signals. The signal handling code then performs the originally intended control flow transfer. Further

control obfuscation techniques were described by László and Kiss [2009], Cappaert and Preneel [2010], Coppens et al. [2013], and Schrittwieser and Katzenbeisser [2011].

Parallelized code. While originally being a code optimization technique, parallelizing code also became popular in the context of code obfuscation, since parallelized code tends to be harder to understand than sequential code [Collberg et al. 1997]. Adding dummy processes to a program or parallelizing sequential code blocks that do not depend on each other [Wolfe et al. 1995] increases the complexity of analysis.

Removing library calls. Calls to libraries of programming languages (particularly ones with a high level of abstraction) offer useful information to an analyst. Because they are called by their name, they cannot be obfuscated. By replacing standard libraries with custom versions, these calls can be removed and thus their functionality obfuscated. Another variant of this obfuscation method is to link libraries statically into the application or to combine many small libraries into a few large ones.

Breaking relations. This technique aims at obfuscating relations between components of a program such as the structure of the call graph or the inheritance structure of an object-oriented program. For example, classes can be split up (*factoring*); similarly, common features of independent classes that do not have common behavior can be moved into a new parent class (*false refactoring*). Sosonkin et al. [2003] presented concepts for Java bytecode obfuscation through class coalescing, class splitting, and object type hiding. A related approach by Sakabe et al. [2005] takes advantage of concepts in object-oriented languages such as polymorphism to obfuscate the relation between objects. Moreover, the idea of class hierarchy flattening to remove all inheritance relations from object-oriented programs was introduced in the literature [Foket et al. 2013, 2014].

3.3. Dynamic Code Rewriting

The main characteristic of code obfuscation schemes in this category is that the executed code differs from the code that is statically visible in the executable.

Packing/Encryption. Various malware obfuscation approaches analyzed in the literature follow the concept of *packing*, which hides malicious code by encoding or encrypting it as data that cannot be interpreted by static analysis. An unpacking routine turns this data back into machine-interpretable code at runtime. By changing the encryption/encoding keys, packed program code can easily be rewritten upon distribution to complicate simple pattern matching analysis (*polymorphism* [Nachenberg 1997]).

The concept of *packing* is also used for benign software. Both the reduction of storage requirements though compression and the aim for obfuscating the code of an application to deter program analysis are key motivations for the adaption of packing technologies. For these application areas, a large number of commercial packers such as VMProtect,[4] ASPack,[5] Armadillo,[6] Execryptor,[7] Enigma,[8] PECompact,[9] and Themida[10] as well as open-source tools (e.g., UPX[11] and Yoda[12]) exist. Most of these tools are also popular

[4]http://vmpsoft.com (accessed February 25, 2015).
[5]http://www.aspack.com (accessed February 25, 2015).
[6]http://www.siliconrealms.com/armadillo.php (accessed February 25, 2015).
[7]http://www.strongbit.com/execryptor.asp (accessed February 25, 2015).
[8]http://enigmaprotector.com (accessed February 25, 2015).
[9]http://bitsum.com/pecompact (accessed February 25, 2015).
[10]http://www.oreans.com/themida.php (accessed February 25, 2015).
[11]http://upx.sourceforge.net (accessed February 25, 2015).
[12]http://yodap.sourceforge.net (accessed February 25, 2015).

with malware authors to hide the maliciousness of their code [Brosch and Morgenstern 2006].

The concept of packing for software protection was also discussed in academia. Cappaert et al. [2006] introduced a modified form of packing, where code can be decrypted at fine granularity right before execution using a key derived from other code sections. Wu et al. [2010] introduced a polymorphism-based concept called *mimimorphism* which encodes data into a representation that looks like program code. A taxonomy of packer-based obfuscation schemes as well as similar techniques was presented by Mavrogiannopoulos et al. [2011]. Recently, Roundy and Miller [2013] presented a survey on obfuscation techniques used in malware packers.

Dynamic code modification. In this technique similar functions are obfuscated by providing a general template in memory that is patched right before its execution [Collberg et al. 1997]. Static analysis techniques fail to analyze the program, as its functionality is available at runtime only. Other concepts of dynamic code modification [Kanzaki et al. 2003; Madou et al. 2006a] implement the idea of correcting intentionally erroneous code at runtime right before execution.

Environmental requirements. Riordan and Schneier [1998] proposed the concept of environmental key generation, in which a cryptographic key is not statically stored in a binary but constructed from *environmental* data collected from within the computing environment. Only if a specific environmental condition is met (called activation environment) is the program able to generate the key and execute its code. Outside the activation environment the program does not reveal its secrets to an analyst.

Similar concepts can be applied to code as well. Sharif et al. [2008] proposed a malware obfuscation scheme that makes the code conditionally dependent on an external trigger value. Without knowledge of this specific value, the triggered behavior is concealed from dynamic analysis. Similar techniques are widely used in malware.

Hardware-assisted code obfuscation. Hardware tokens can be used to improve the strength of other code obfuscation techniques [Fu et al. 2007; Zhuang et al. 2004; Bitansky et al. 2011]. The basic idea is to create a hardware-software binding by making the execution of the software dependent on some hardware token. Without this token, analysis of the software will fail, because important information (e.g., targets of indirect jumps) is not available.

Hardware-based isolation mechanisms for trusted computation such as Intel SGX (Software Guard Extensions) allow an application to prevent other applications and even the operating system kernel from accessing certain memory regions [Anati et al. 2013]. Such mechanisms seem well suited to protect code and data from runtime inspection and tampering and may in the future become an impediment to dynamic analysis.

Virtualization. Virtualization describes the concept of converting the program's functionality into byte code for a custom virtual machine (VM) interpreter that is bundled with the program [King and Chen 2006; Ghosh et al. 2010]. Virtualization can also be combined with *polymorphism* by implementing custom virtual machine interpreters and payloads for each instance of the program [Anckaert et al. 2006]. Vrba et al. [2010] proposed the combination of fine-granular encryption and virtualization to hide VM code from analysis. Collberg et al. [1997] described a variant of this concept under the term *table interpretation*. A similar concept by Monden et al. [2004] uses a finite state machine-based interpreter to dynamically map between instructions and their semantics.

Antidebugging and -disassembly techniques. This obfuscation category includes techniques that actively oppose analysis attempts via disassembly or debugging. For

example, attached debuggers can be detected based on timing and latency analysis or the identification of code modifications caused by software breakpoints. Another technique is the execution of undocumented instructions in order to confuse a code analysis tool or a human analyst [Brand 2010].

4. CODE ANALYSIS

We consider four broad classes of analysis against programs protected by obfuscation: pattern matching, automated static analysis, automated dynamic analysis, and manual reverse engineering by a human who has access to automated tools.

Note that there is no universally accepted definition for static or dynamic analysis; in fact, we believe that it is impossible to draw strict boundaries between any of the analysis classes, so we apply our best judgment to arrive at a meaningful classification of existing approaches to analyzing programs. Madou et al. [2005] argue that considering resilience against only particular types of analysis can leave a technique open to circumvention by hybrid approaches.

4.1. Pattern Matching

With the term "pattern matching," we refer to fast automated techniques for finding and unveiling known structures in binary programs. For instance, this could be artifacts introduced by obfuscation tools or signatures of well-known libraries and code samples. In contrast to our definition of static and dynamic analysis, pattern matching is purely syntactic and does not reason about the program semantics. The types of patterns to look for can range from simple byte strings to regular expressions or other languages allowing wildcards. We also consider machine learning techniques that essentially treat programs as byte strings to be pattern matching.

One example for pattern matching is the Fast Library Identification and Recognition Technology (F.L.I.R.T.) in IDA.[13] It uses signatures to identify library functions with the goal of simplifying reverse engineering. Function signatures of well-known libraries are stored in a database; when disassembling a new binary, its bytes are checked against known signatures. Recognized functions are named and annotated using the stored information to aid a human reverse engineer in understanding the functionality of the code.

Usually, the patterns to look for have been generated ahead of time or are curated by human analysts. Differential pattern matching approaches work without a priori knowledge and instead compare static or dynamic artifacts, such as control flow graphs [Flake 2004; Nagarajan et al. 2007] or instruction traces [Zhang and Gupta 2005]. As mentioned when discussing our methodology in Section 2.1, we consider this type of pattern matching a multistage analysis with an initial static or dynamic phase. Note that a simple translation of byte patterns to instruction opcodes in the manner of a linear sweep disassembler would still fall under pure pattern matching; we draw the boundary to static analysis where a disassembler interprets possible jump targets.

4.2. Static Analysis

Static analysis is widely used for optimizing code, finding or proving the absence of bugs, and reverse engineering. In its broadest sense, static analysis refers to any program analysis that is performed just by inspecting executable code or a disassembled representation of a program of interest but without ever executing it on a real or virtual machine. In this survey, we distinguish static analysis from syntactic pattern matching as describing analyses that reason about the program semantics. For a detailed

[13]https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml (accessed February 25, 2015).

introduction to static program analysis, we refer the reader to the classic textbook by Nielson et al. [1999].

Static analysis deduces information about a program by reasoning about the possible executions of the target program. A static analysis is considered to be *sound* if it is guaranteed to include *all* possible executions of a program in its judgment. For example, a static analysis that attempts to generate a control flow graph from a binary program is sound if the resulting graph contains at least all those control transfers that will ever occur at runtime. Because of undecidability, static analysis can only achieve soundness by *overapproximating*, that is, by generalizing the concrete program behavior and accepting that this generalization will also include behavior that does not occur in real executions. To continue the example, a sound control flow graph could contain edges that will never be taken at runtime. Overapproximation is the reason why static analysis can report false positives when it is used for bug finding or malware detection.

In principle, the precision of a static analysis can be increased to reduce the number of false positives, but such added precision comes with a corresponding increase in cost. As a compromise, many practical static analysis systems choose to be *unsound*, that is, they do not guarantee to cover all possible behaviors. This may introduce false negatives in addition to false positives but is often an acceptable tradeoff to make systems useful in practice.

An example of the tradeoffs in precision, performance, and soundness can be seen in the various approaches to reconstruct the control flow of binaries: a linear sweep disassembler that disassembles byte after byte starting at the entry point performs a low-precision static analysis that is unsound because it can miss code reached through branches [Schwarz et al. 2002] (in fact, since its interpretation of the instruction semantics is trivial, we classify linear sweep as pattern matching, see above). A recursive traversal disassembler improves accuracy by trying to identify and follow the location of branches during analysis, but it will still leave out many indirect branches whose targets are computed at runtime. Disassembly by abstract interpretation performs a sound static analysis of the binary code to also systematically resolve indirect branches whose targets are computed at runtime [Kinder et al. 2009]. The computational cost depends on the exact type of analysis performed but can be significant.

Analysis platforms such as CodeSurfer/x86 [Balakrishnan and Reps 2004], Jakstab [Kinder and Veith 2008], or Bap [Brumley et al. 2011] provide the necessary infrastructure and abstractions to apply higher-level static analysis to binaries. Typically, low-level details of instruction encoding and semantics are abstracted by such platforms, and custom analyses can work on a simplified low-level language instead of machine code.

From a conceptual standpoint, obfuscation decreases the precision a particular static analysis can achieve, that is, obfuscation introduces additional sources of overapproximation [Giacobazzi and Mastroeni 2012]. Recall from Section 3.2 that control flow flattening forces all control flow to pass through a dispatcher. Because the dispatcher is executed many times on all paths, a static analyzer that generalizes the program behavior at the location of the dispatcher will be very imprecise. To maintain precision, a static analyzer can compensate by generalizing the behavior at a finer granularity [Kinder 2012].

In general, whenever the details of a particular obfuscation technique are known, an analysis can be crafted to almost completely eliminate the effect of the obfuscation [Krügel et al. 2004]. However, despite existing work on automated refinement of the precision of general static analysis [Das et al. 2002; Bardin et al. 2011], tailoring analyzers to cope with obfuscation schemes is still a largely manual process.

Besides the conceptual precision requirements, there are also significant practical challenges for statically analyzing obfuscated code. All existing tools make some

assumptions about the behavior or structure of the executable code, which can be broken by obfuscations. Many static analyzers fail even in the presence of simple obfuscations used regularly by malware [Moser et al. 2007b]. Especially tools that depend on an initial disassembly phase, such as CodeSurfer/x86 [Balakrishnan and Reps 2004] or early semantic malware detectors [Christodorescu et al. 2005; Kinder et al. 2005], are vulnerable to syntactic obfuscations. Such obfuscations target disassemblers that rely mostly on heuristics to discover all executable code [Linn and Debray 2003]. Removing the separate disassembly phase and working on code directly improves resilience against these simpler obfuscation schemes [Kinder et al. 2009; Brumley et al. 2011]. Nevertheless, especially dynamic obfuscations are difficult to handle for static analyzers. For example, no static tool to date is able to reliably deal with self-modifying code, even though one could, in theory, imagine the static analysis compensating by analyzing the possible runtime state of the code section.

However, static analysis of obfuscated code can still give a reverse engineer valuable information and lead to a better understanding of the overall binary and further guide additional deobfuscation steps. Our analysis in Section 5 considers static analysis tools at the current state of the art, including minor adaptations for each particular scenario.

4.3. Dynamic Analysis

Dynamic analysis refers to observing the behavior of deployed and running systems, and it is today an important part of forensic analysis of malware [Egele et al. 2012]. It analyzes real executions of a program, either online (at runtime) or offline (over a recorded trace). In particular, any form of software testing is a dynamic analysis. Just like static analysis, dynamic analysis is also performed with respect to particular properties of interest. For example, a dynamic analysis may record system call invocations or executed instructions or trace the flow of "tainted" data that is received from the network.

Dynamic analysis is conceptually dual to static analysis: A sound dynamic analysis considers a subset of all execution traces of a program and is therefore *underapproximate*. Each bug, warning, or suspicious behavior found is then guaranteed to also occur during at least one real execution. On the flip side, a dynamic analysis may miss behavior when the number of possible traces in a program is too large to be exhaustively tested. In the general case, this is unavoidable due to undecidability.

While static analysis trades off precision against cost, dynamic analysis trades off *coverage* against cost. Each execution of the program under test corresponds to an additional trace covered. An automated dynamic analysis will typically try to cover traces that exhibit diverse behaviors to get a representative profile of the program. If a program exhibits a particular type of behavior only under very specific circumstances, then it may never be observed. This is especially problematic for malware, which may respond only to certain "triggers" [Crandall et al. 2006; Kolbitsch et al. 2011]. This challenge is closely related to the problem of automated test case generation, and approaches can be roughly classified into blackbox, graybox, and whitebox depending on how they treat the program under test.

Blackbox testing assumes no knowledge of the program and is usually done via random input generation. Since it is very hard to achieve meaningful coverage through blind enumeration of inputs, effective fuzz testing tools employ a *graybox* approach and use domain-specific knowledge about the input format to generate inputs that have different meaning (e.g., image files with different sizes). Coverage-guided fuzzers such as AFL[14] mutate an initial input following a variety of heuristics. *Whitebox* testing also uses the program structure to guide input generation. Tools based on symbolic

[14]http://lcamtuf.coredump.cx/afl/ (accessed June 07, 2015).

execution [King 1976] and dynamic test generation (also referred to as concolic execution in the literature) have been particularly successful [Godefroid et al. 2005; Cadar et al. 2006; Sen et al. 2005]. They use a constraint solver to generate inputs systematically, such that one input will cover exactly one unique control flow path through the program. However, since the number of control flow paths can be infinite (due to loops), even symbolic execution will remain incomplete and thus underapproximate in a finite amount of time. A survey by Schwartz et al. [2010] provides a good introduction to symbolic execution and explains the related challenges in more detail.

In practice, systematic exploration methods such as symbolic execution are additionally limited by their underlying constraint solver which is used to identify valid control flow paths. A new control flow path that exhibits hitherto unseen behavior can only be triggered if the constraint solver is able to find an input that drives execution down that path. If the constraint is unsupported (e.g., floating point) or simply too difficult, then the symbolic execution engine is unable to cover the path. As a best effort solution, the engine can then resort to random testing (fuzzing) of input parameters [Godefroid et al. 2008]. As with static analysis, there are also dynamic techniques that compromise on soundness. For instance, it is possible to directly manipulate registers or memory to trigger different branches or execute particular areas of code [Madou et al. 2005; Moser et al. 2007a]. While this may put the program into a state that would never occur under normal circumstances, it will cause a particular path to be executed without requiring any constraint solving, which may just be enough for the particular reverse engineering task at hand.

A significant practical advantage of dynamic over static approaches in the analysis of obfuscated code is that it can be applied to binaries with relative ease. In fact, it is often simpler to dynamically analyze binaries than source code, because traces recorded at runtime show addresses of instructions in the binary and not just (obfuscated) source code information. However, the use of antidebugging techniques can oppose dynamic analysis attempts.

Because dynamic analysis monitors what is actually executed at runtime, obfuscations cannot fully conceal the behavior of a program. For instance, a dynamic analysis can trace self-modifying code just like regular code if it records the opcode and operands of the current instruction in addition to the value of the program counter [Thakur et al. 2010].

As for static analysis, Section 5 lists what is possible using the current state of dynamic analysis tools, with only relatively minor technical adaptations.

4.4. Human-Assisted Reverse Engineering

We use this very broad category to cover any kind of analysis that a skilled human reverse engineer can perform with the help of any state-of-the-art tool. The ability for creative problem solving and adaptation makes humans much more efficient in dealing with obfuscations than fully automated techniques. In contrast to a purely automated approach, however, humans can be misled by clues that suggest structure, such as type names or inheritance relationships among classes.

Disassemblers and decompilers alike have gone through major improvements over the past two decades [Cifuentes and Gough 1995; Schwarz et al. 2002; Schwartz et al. 2013]. The de facto industry standard for disassembling and reverse engineering, IDA Pro, now includes a powerful decompiler for the x86, x64, and ARM processor architectures.[15] Academic research has made significant progress as well. The Boomerang Project[16] is an attempt to develop an open-source decompiler. Early results from

[15]https://www.hex-rays.com/products/decompiler (accessed February 25, 2015).
[16]http://boomerang.sourceforge.net (accessed February 25, 2015).

a study [Emmerik and Waddington 2004] conducted on a real-world program were promising. Despite the fact that only the core algorithm and not the entire program was decompiled, the experiment was able to demonstrate that decompilation with human assistance can be practical in certain use cases. The project, however, does not seem to receive much attention anymore. Recently, a new binary-to-C decompiler named PHOENIX was introduced by Schwartz et al. [2013]. It implements a structural analysis algorithm that uses iterative refinement strategies as well as the property of semantics preservation in order to archive significantly more accurate results than previous approaches. The reconstruction of high-level control flow structures is a challenging problem in decompiler research. State-of-the-art decompilers rely on structural analysis and advanced pattern matching and fall back to using `goto` statements for control flow transfers if no patterns can be identified. Recently, Yakdan et al. [2015] proposed a decompiler that does not rely on pattern matching and produces more compact and `goto`-free code. It uses semantics-preserving transformations to restore structured control flow graphs.

A major driving force behind the development of human-assisted code analysis approaches is the software cracking scene. In early years, the SoftICE debugger was a popular tool among reverse engineers; however, it is no longer maintained [Willems and Freiling 2012]. Today, the freely available debugger OllyDBG[17] to some extent continues where SoftICE left off. A large number of plugins that are dedicated to cracking and tampering purposes have been made available by the scene (e.g., Ollybone [Stewart 2006] for semiautomatic unpacking).

The concepts and tools in this code analysis category have in common that the automated analysis process can be directly influenced by the human reverse engineer resulting in very different outcomes depending on the human's decisions. For example, the disassembling process of IDA Pro can be influenced by a human reverse engineer to perform recursive traversal at specific locations of the program's code. This code analysis category is naturally hard to grasp formally. In our analysis in the next section, we base the capabilities of human reverse engineers on published results and common knowledge about the state of the art. A comprehensive introduction to software reverse engineering was published by Eilam [2005].

5. ROBUSTNESS ANALYSIS

In this section, we evaluate the effectiveness of different classes of software obfuscation schemes against the program analysis scenarios introduced in Section 2.4. The effectiveness of a specific type of code obfuscation is evaluated by comparing it to code analysis approaches described in the literature. We are well aware of the arms race between code obfuscation and analysis and the fact that in theory every obfuscation technique can be broken with targeted analysis techniques (see Section 4). In this survey, we focus on the status quo of code obfuscation in real-life application scenarios and evaluate the capabilities of state-of-the-art code analysis tools (also considering possible non-complex modifications to the analysis techniques) in order to target particular obfuscations. The possibility of developing analysis techniques targeted to break a specific obfuscation scheme does not prove it useless in general, as a small modification to the obfuscation technique can again raise the bar for analysis.

As of today's knowledge, a precise formalization of the security of an obfuscation scheme seems to be difficult to achieve. Previous attempts to quantify the hardness of a particular class of code obfuscation are based on software complexity metrics [Collberg et al. 1997; Anckaert et al. 2007]. However, it remains unclear whether such notions are able to capture all security properties of the obfuscating transformation correctly.

[17]http://www.ollydbg.de (accessed February 25, 2015).

Table II. Analysis of the Strength of Code Obfuscation Classes in Different Analysis Scenarios

	PM		Autom. Static				Autom. Dynamic				Human Assisted			
Name	**LD**	**LC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**
Data obfuscation														
Reordering data	■	▒	✓			▒	✓							
Changing encodings	✓	▒	■			▒	✓							
Converting static data to procedures	✓	▒	■			▒	✓				▒			
Static code rewriting														
Replacing instructions	■	✓		✓		▒								
Opaque predicates	■	▒	▒	✓	▒	▒			▒	▒			▒	
Inserting dead code	■	▒				✓			✓	✓				
Inserting irrelevant code	■	✓				▒								
Reordering	■	▒				▒								
Loop transformations	■	▒	▒	▒	▒	▒			▒	▒				
Function splitting/recombination	■	▒				✓								
Aliasing	■	▒	▒	▒	■	✓			✓	▒			▒	
Control flow obfuscation	✓	▒	▒	▒	■	▒			✓	✓		✓		✓
Parallelized code	■	▒	▒	▒	■	■			▒	▒				
Name scrambling	■	▒				✓						▒	▒	▒
Removing standard library calls	■	▒	▒	▒	▒	▒			▒	▒		▒	▒	✓
Breaking relations	■	▒			▒	▒			▒	▒		▒	▒	▒
Dynamic code rewriting														
Packing/Encryption	■	✓	■	✓	■	■		✓	✓			✓		✓
Dynamic code modifications	■	■	■	■	■	■		▒	▒	▒				
Environmental requirements	■	■	■	■	■	■	▒	▒	▒	▒	▒	▒	▒	▒
Hardware-assisted code obfuscation	■	■	■	■	■	■	▒	▒	▒	▒	✓	▒	▒	▒
Virtualization	■	■	✓	✓	■	■		✓	▒	▒		✓	▒	✓
Anti-debugging techniques			▒	▒	▒	✓	■	✓	✓	■	▒	▒	▒	✓

Legend		
	■	obfuscation breaks analysis fundamentally
	▒	obfuscation is not unbreakable, but makes analysis more expensive
		obfuscation only results in minor increases of costs for analysis
	✓	A checkmark indicates that the rating is supported by results in the literature
		Scenarios without a checkmark were classified based on theoretical evaluation

PM = Pattern Matching, LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code.

In this survey, we thus follow a different approach and rate the strength of each obfuscation class. A summary of the results can be found in Table II. Black marks obfuscation techniques that break a certain type of program analysis fundamentally with respect to its state-of-the-art techniques. Gray denotes scenarios in which a particular code obfuscation class cannot be considered unbreakable but still makes analysis substantially more expensive. For example, while an analysis technique might work under lab conditions when dealing with toy programs, limits of available resources could be reached while analyzing larger programs. White marks obfuscation techniques that

only slightly raise the cost of analysis and therefore cannot justify the cost of adopting them, such as increased code size or added runtime overhead. Furthermore, we distinguish between ratings that are supported by results in the literature (marked with a checkmark in Table II) and ones that we based on our own evaluation of the state of the art.

5.1. Pattern Matching

The most basic form of pattern matching is limited to comparing program code on byte level against some predefined pattern. Consequently, this type of pattern matching is weak against all kinds of code modifications, including obfuscating transformations. However, several more sophisticated pattern matching concepts, mostly for identifying malicious behavior, were introduced in the literature. Such advanced pattern matching arguably can cope with naive obfuscation techniques such as equivalent instructions. In the following, we discuss the state of the art in research on pattern matching-based code analysis in the presence of different classes of code obfuscation schemes.

Locating data. Naïve pattern matching techniques on data break as soon as the structure of the data changes. Thus, even simple obfuscations are effective against simple forms of pattern matching. This was confirmed by Moser et al. [2007b], who were able to show that simple data obfuscation techniques are sufficient to make pattern matching ineffective for the identification of data. *Antidisassembly techniques* are not effective against pattern matching-based analysis of the binary, since they do not necessarily require the program to be disassembled.

Locating code. Moser et al. [2007b] also demonstrated the limitations of pattern matching against the static code rewriting technique *control flow flattening*. Their reasoning can be easily extended to other forms of static rewriting and dynamic code rewriting as any modification of the binary clearly destroys the pattern.

In the literature, locating code through pattern matching is often described in a malware context. The primary aim of techniques in this analysis scenario is the generation of generic patterns describing malicious behavior of program fragments to automatically classify software as malicious or benign. While these approaches do not aim at understanding the semantics of the program, they still can identify locations in the code that implement abnormal (malicious) behavior. HANCOCK [Griffin et al. 2009] is a system for the automated generation of signatures for malware. Through normalization of opcodes, it is resistant against simple code transformations such as register reassignment. Dalla Preda et al. [2011] generated signatures of metamorphic malware through abstract interpretation of semantics developed for self-modifying code. Generally, the expressiveness of the language used for creating patterns (e.g., static strings vs. regular expressions) dictates both their resilience against obfuscations and the performance of matching algorithms. Since existing techniques emphasize performance over expressiveness, we marked static code rewriting techniques in gray.

Dynamic code obfuscation techniques, for instance, virtualization, remove the structure of the code entirely, thus rendering pattern matching-based analysis approaches ineffective. To some extent, however, code analysis based on patterns is still possible as demonstrated in the literature. Tang and Chen [2007] proposed the identification of polymorphic malware on a network stream using statistical analysis. In contrast to fixed string pattern matching, polymorphic versions of a program can be identified with this approach. POLYGRAPH [Newsome et al. 2005] is a concept for the automated identification of polymorphic worms which exploits the existence of invariant substrings in all polymorphic variants of a malware. Both approaches exploit the characteristic structure of polymorphic programs. To summarize, while it was shown in literature

that pure identification of the obfuscation method *polymorphism* is feasible in a malware context, locating particular functionality inside the binary is still impossible with pattern matching if the implementation details are unknown to the analyst. As in the previous scenario, *antidisassembly techniques* do not provide additional protection against pattern matching approaches aiming at locating code.

5.2. Static Code Analysis

Locating data. Similarly to pattern matching, automated static analysis of data is limited when analyzed data are not stored in its original representation. In Table II, the data obfuscation class *reordering* is marked in gray, because a simple dataflow analysis, which is necessary for the reconstruction of reordered data, is arguably possible with automated static analysis tools such as JAKSTAB [Kinder and Veith 2008]. Furthermore, it can be argued that a static code rewriting technique can be effective against locating data if it complicates the reconstruction of the control flow graph of the program: Static dataflow analysis strongly depends on knowledge of the control flow. For this reason, control flow modifying obfuscation techniques are marked in gray in Table II.

Dynamic code rewriting in general is effective against locating data inside programs as the original representation of the data is destroyed. At first sight, static analysis appears completely helpless against virtualization obfuscation, as only the code of the interpreter can be directly analyzed. In static analysis, this leads to an effect that Kinder [2012] called *domain flattening*: Dataflow information from different locations in the original program are merged to one location in the interpreter, resulting in a more imprecise analysis. However, Kinder [2012] was able to demonstrate that by lifting the static analysis to a second dimension of location (the virtual program counter), the same analysis precision as on nonobfuscated code is achievable. While the introduced approach was evaluated on a toy example only, the preliminary results still indicate promising directions for static analysis of VM-protected binaries. For that reason, we marked *virtualization* in gray for this analysis scenario. *Antidisassembly techniques* can, to some extent, limit static analysis and thus are marked in gray as well.

Locating code. Locating a particular program feature in binary code through static analysis can be based on an analysis of the structure of the control flow graph (e.g., Harris and Miller [2005]) of the program. Therefore, code obfuscation schemes that modify or hide the control flow of a program (*opaque predicates*, *loop transformations*, *parallelized code*, etc.) can be considered as candidates for protection against static analysis techniques. *Opaque predicates* are the most important concept for control flow obfuscation against static analysis tools. Dalla Preda et al. [2006] proposed an abstract interpretation-based methodology for removing simple *opaque predicates*. This automated, static concept was shown to be more complete than approaches for dynamic analysis of opaque predicates. However, as the authors state, their analysis concept is limited to simple types of opaque predicates only. Thus, we consider more complex opaque predicates still effective in making static reconstruction of the control flow graph significantly more difficult.

Tools for matching program code based on control flow similarities include BINDIFF [Flake 2004], BINHUNT [Gao et al. 2008], BINSLAYER [Bourquin et al. 2013], and the implementation by Nagarajan et al. [2007]. Moreover, Tsai et al. [2009] introduced a framework for analyzing control flow obfuscation by representing it as a composition of atomic operators to evaluate robustness. Still, it remains unclear to what extent such theoretical results can support the evaluation of the strength of an obfuscation in real-life applications. We conclude that while no general statement regarding the strength of obfuscation can be made for this particular analysis scenario, obfuscation

schemes that complicate the reconstruction of the control flow graph can still make the analyst's aim of identifying the entry point into a particular functionality considerably harder.

Code obfuscation based on dynamic code rewriting makes static analysis considerably more difficult as the analyzed code in the binary does not correspond to the code that is actually executed. However, approaches to circumvent obfuscation were introduced in the literature on malware samples. *Malware packers* were analyzed with heuristics-based static analysis techniques (e.g., Treadwell and Zhou [2009]) and comparison with previously seen malware samples was performed [Jacob et al. 2012; Karnik et al. 2007]. Similarly to pattern-matching-based approaches, in automated static analysis, the maliciousness of code is evaluated by identification of abnormal structures of the program code. The idea of using model checking for detecting malicious code was proposed by Kinder et al. [2005]. Furthermore, static analysis of *polymorphism* as well as *metamorphism* was discussed in recent literature. Bruschi et al. [2006a] compared a normalized version of the control flow graph of a binary against control flow graphs of known malware in order to detect malicious behavior. In a similar concept, Walenstein et al. [2006] normalized program code (e.g., through simple instruction substitution patterns) to be able to compare it to known malware samples. Another code normalization technique was presented by Bruschi et al. [2006b]. Coogan et al. [2009] used a combination of two static code analysis techniques (slicing and alias analysis) to identify malware packers. The idea of using the frequency of opcodes as a predictor for malware was proposed by Bilar [2007]. For example, massive use of mathematical operations might indicate malware that tries to obfuscate its malicious behavior using a packer-based approach. Furthermore, it was shown that a high frequency of a usually rare opcode is an indicator for polymorphic or metamorphic malware. Chouchane and Lakhotia [2006] proposed the detection of metamorphic malware by creating signatures for instruction-substitution engines. However, similar to pattern-matching-based approaches, none of the introduced concepts for static analysis is able to find the location of specific functionality. Only the characteristic structure of a malware packer can be identified. As a consequence, we marked *packing/encryption* and *replacing instructions* in gray. Like in the previous scenario (locating data), *virtualization obfuscation* was marked gray in Table II due to preliminary results on the challenges of statically analyzing virtualization-obfuscated programs by Kinder [2012].

Extracting code. While the software engineering literature knows of concepts for reusing functionality from legacy binary code (e.g., Sneed [2000]), the automated static extraction of obfuscated code has not been widely discussed.

Our reasoning regarding the effectiveness of different classes of code obfuscation techniques in this analysis scenario is simple: Extracting code from a program through static analysis is at least as difficult as locating code, because the latter is a fundamental requirement for the former. Table II indicates differences between locating and extracting code in the scenario of static code rewriting for four obfuscation techniques. *Aliasing*, *control flow flattening*, *parallelized code*, and *breaking relations* share the common effect of increasing code dependencies by interweaving independent parts of the program. These interweavings are difficult to resolve through static analysis, thus making the extraction of code sections considerably harder than just locating code. Collberg et al. [1997] first described the effect of raising analysis complexity when extending the scope of the obfuscating transformation. Analogously, in the category of dynamic code rewriting, *virtualization* was marked in black because of the interweavings that make it difficult to extract all required code sections. *Antidebugging and Antidisassembly* was marked in black due to the existence of a wide range of techniques that actively interfere with static disassembly [Branco et al. 2012]. This

includes concepts to prevent a disassembler from recovering a correct higher level representation of the code [Popov et al. 2007].

Understanding Code. For a human analyst to better understand an obfuscated program, an automated analysis tool has to be able to remove at least parts of the applied code obfuscation scheme from the binary.

Several concepts for static deobfuscation with the aim for making the program code more understandable for a human analyst were proposed in the literature. An early work by Rugaber et al. [1995] has shown that detecting interleaved code (e.g., through *function recombination*) is a time-consuming task. Majumdar et al. [2006] evaluated the robustness of the obfuscation technique *aliasing*, where two or more pointers refer to the same memory location. An experimental evaluation showed that resilience expected from the theoretical approach does not hold in real-life scenarios; still, in general it is difficult to evaluate the actual strength of aliasing. Guillot and Gazet [2010] developed techniques for automated static deobfuscation. The basic concept is to make program code more understandable by automated rewriting based on local semantic analysis, similarly to optimization steps made by compilers. Raber and Laspe [2007] introduced a plugin for IDA Pro that is capable of removing basic obfuscation and antidebugging techniques from a binary. A shared limitation of all introduced concepts is the major gap in success rates between academic examples and real-world scenarios. While these concepts show that theoretical analysis models work under laboratory conditions, practical application is limited. Thus, we marked data obfuscation as well as static code rewriting techniques in gray.

In Table II *name scrambling* was marked in black. Identifier names are often critical to human understanding of a program but cannot be fully restored with the help of automated code analysis techniques. Collberg et al. [1997] described this transformation as one-way, although limited recovery is possible via code analysis (e.g., induction variables of loops can be identified and renamed to "i", dynamic data structures can be renamed to common identifiers, etc.). *Parallelized code* can implicitly be considered as a strong obfuscation technique in this analysis scenario as code extraction, which is a fundamental requirement for code understanding, and is also difficult. Following the same line of reasoning, dynamic code rewriting methods in general are effective against code understanding through automated static analysis.

5.3. Dynamic Code Analysis

Locating data. Every known data obfuscation technique except for whitebox cryptography has a limited practical effect in dynamic analysis scenarios since data is always visible at some point during runtime. For example, although the cryptographic key of a DRM client might be stored in an obfuscated way (e.g., through data encoding) in the binary, during runtime the cryptographic algorithm has to reconstruct the original representation of the key in order to perform decryption tasks. In the literature, several concepts for an automated dynamic extraction of data structures from program binaries were introduced. Shamir and Van Someren [1999] proposed the identification of cryptographic keys in binary code through entropy analysis—a concept which can also be applied to memory.

Zhao et al. [2011] introduced a concept for dynamic extraction of data in malware. Cozzie et al. [2008] extracted data structures from memory dumps using Bayesian unsupervised learning. Lin et al. [2010] introduced a system called REWARDS which reveals data structures through observation of the program execution. It marks each memory location that was accessed at runtime with a timestamp and traces the propagation of data. A similar concept called HOWARD was proposed by Slowinska et al. [2011] in 2011. It aims at extracting data structures from binaries by dynamically tracing

(using QEMU-based emulation) how a program accesses the memory. HOWARD is able to reconstruct large parts of the symbol table. It thus simplifies the progress of reverse engineering and improves readability of obfuscated code as well as data.

To sum up, in a dynamic analysis context, most static as well as dynamic code rewriting techniques do not provide significant additional security in the context of data protection. The only exception are obfuscations that require special runtime enablers (additional hardware or environmental conditions) to execute. These techniques can withstand dynamic analysis in situations where the runtime enabler is not present. For this reason, both the techniques *environmental requirements* and *hardware-assisted code obfuscation* were marked in gray in Table II.

Locating code. Locating a particular feature inside binary code through dynamic analysis is based on the observation of the program behavior. Static code rewriting techniques are not effective in dynamic analysis scenarios due to one important factor: Most of them are not explicitly targeted at the prevention of dynamic analysis. *Replacing instructions*, *inserting dead code*, *opaque predicates*, *code insertions*, and so on, were in the first place developed to obfuscate the static representation of binary code. However, automated dynamic analysis techniques do not depend on the static representation as much as static analysis or even human analysis do; thus they are less affected by the obfuscation. In the literature, very diverse concepts for dynamic analysis of obfuscated code were proposed. Li et al. [2009] described a technique that identifies malicious behavior based on the malware's runtime system call sequences. MCVETO [Thakur et al. 2010] is a dynamic test generator and model checker for machine code. While traditional dynamic analysis suffers from the problem of incompleteness as a program behavior can only be analyzed on one input at at time, MCVETO implements a combined static and dynamic approach which aims at reaching more code locations by actively manipulating program inputs. Another strategy for finding a particular feature in program code was discussed by Deprez and Lakhotia [2000] and Wilde and Scully [2006]. The basic idea is to execute the program twice with two different inputs whereby one input invokes the feature and the other does not. From calculating the differences between the two traces conclusions on the location of the particular feature can be drawn. Madou et al. [2005] discussed the effectiveness of hybrid (static and dynamic) analysis approaches and demonstrated it in the context of the reconstruction of an obfuscated control flow graph. Zhang and Gupta [2005] proposed the comparison of instrumented executions of different program versions for matching purposes. Brumley et al. [2008] introduced a concept for automated generation of exploits based on static and dynamic comparison of programs and patched versions of it.

While dynamic code analysis is strong against static code obfuscation, dynamic obfuscation techniques are much more robust because executed code differs from the code that is statically visible in the executable. However, analyses targeting dynamic code rewriting obfuscations were presented in the context of malware analysis, in particular for packed programs. Moser et al. [2007a] proposed a solution for the incompleteness problem of dynamic analysis by making the exploration of multiple execution paths possible. Thus, the approach allows the identification of malicious behavior that is executed only if a certain conditions is met. REANIMATOR [Comparetti et al. 2010] is a two-step malware identification system. First, known malware is dynamically analyzed and code that is responsible for a particular malicious behavior is modeled. This model can then be used in a second step to identify the same malicious behavior in other code samples through static analysis. However, dynamic code rewriting techniques such as packing can limit the detection rate of this approach significantly. Sharif et al. [2008] introduced the EUREKA framework that automatically extracts the payload of a packed program by running the binary in a virtual machine. RENOVO by Kang et al. [2007] is

another dynamic unpacker that monitors executed instructions and memory writes at runtime to extract a hidden payload and is based on the dynamic code analysis component TEMU [Yin and Song 2010] of the BITBLAZE platform [Song et al. 2008]. Heuristics- and statistics-based strategies are used to determine the exact moment when the unpacking process is finished and the unpacked code is fully stored in memory. Royal et al. [2006] introduced a behavioral-based approach for automated unpacking inside a VM. Its combination of static and dynamic analysis identifies unpacking routines through its characteristic behavior. A malware analysis approach by Debray and Patel [2010] focuses on the automated identification of unpacking routines inside binaries. Gröbert et al. [2011] proposed the detection of cryptographic algorithms by analyzing program execution traces, which show unique characteristics depending on the implemented algorithm. In a similar concept named ALIGOT [Calvet et al. 2012], the identification of cryptographic algorithms in execution traces is based on the comparison of input-output relationships with known cryptographic algorithms. With this concept, even heavily obfuscated algorithms can be identified because the input-output relationship does not differ from the original version of the algorithm.

Furthermore, several approaches for automated dynamic analysis of programs protected by virtualization were introduced in recent years. Sharif et al. [2009] proposed the use of taint-flow and dataflow analysis techniques to find the byte code implementing the payload of the virtualized program. The described automated reverse engineering approach is able to reconstruct control flow graphs and was evaluated against the code virtualization tools VMProtect and Code Virtualizer. A different strategy for automated dynamic analysis of virtualized code was proposed by Coogan et al. [2011]. Instructions that contribute to arguments of system calls are collected to understand the functionality of the program. Webster and Malcolm [2009] proposed the use of formal algebraic specifications to detect metamorphic and virtualization-based malware. TRACESURFER [Guizani et al. 2009] uses dynamic binary instrumentation for the detection of self-modifying malware.

To conclude, we marked dynamic code rewriting approaches in gray because practical application of all described approaches is limited to malware identification tools only. In other words, they do not directly aim at locating particular functionality but malicious behavior in general.

Extracting code. Similarly to the scenario of static extraction of code sections, dynamic code extractors have to deal with dependencies between different parts of the program. Several static classes of code obfuscation add bogus dependencies in order to make analysis more difficult. Thus, we can make similar assumptions on the resilience of the code obfuscation techniques. Still, several dynamic concepts for automated extraction of code section were described in the literature. TOP by Zeng et al. [2013] collects instruction traces and translates the executed instructions into a high-level program representation that can be reused as a normal C function in new software. The authors claim the concept to be resilient against the obfuscation techniques *packing/encryption*, *aliasing*, *control flow obfuscation* (e.g., *flattening*), *inserting dead code*, as well as several popular *antidebugging techniques* such as "Soft Breakpoint Detection," "Anti-VMware IN Instruction," and "IsDebuggerPresent Check." Following the results of Zeng et al. [2013], we also marked *parallelized code*, which is arguably less effective in dynamic code analysis scenarios, in gray.

Most other concepts introduced in recent literature are focused on malware. Leder et al. [2009] proposed a concept for automated extraction of cryptographic routines through dynamic data analysis. The automated isolation of a single function from a (malicious) binary was proposed by Caballero et al. [2010]. In contrast, INSPECTOR by Kolbitsch et al. [2010] allows the automated extraction of a particular malicious

behavior that does not necessarily have to be limited to one function of the program only. Following the evaluation of this approach, the dynamic obfuscation class of *packing/encryption* has to be considered weak against the described approach.

Other dynamic code rewriting techniques were marked in gray in Table II. In contrast to the locating code scenario we marked *virtualization* in black as the technique by Sharif et al. [2009] for analyzing virtualization-protected code is limited to locating code and does not yield valid executables.

Understanding code. Analogously to previous scenarios, a deeper understanding of the code of a program requires at least basic deobfuscation in the automated analysis. In literature, several concepts were introduced. Udupa et al. [2005] proposed automated deobfuscation of *control flow flattening* and *dead code* insertion using a hybrid approach of static and dynamic analysis techniques. The incomplete control flow graph from a dynamic analysis is enriched by adding some control flow edges that could possibly be taken through static analysis. While the results presented in the article show that the evaluation of isolated analysis problems is possible, it is difficult to reason about the value of the concept for real-life programs.

5.4. Human Analysis

The capabilities of a human code analyst are difficult to quantify in general. Tilley et al. [1996] first described a framework for program understanding including cognitive aspects of a human reverse engineer. However, it is still safe to assume that provided with sufficient patience a human analyst can break any class of software-only code obfuscation (i.e., an obfuscation where no trust anchor in hardware exists) and is only limited by scalability constraints.

Locating data. Most data obfuscation techniques have only limited strength in the analysis scenario of a human analyst trying to locate data structures in programs. One important concept for the protection of data (keys) inside a binary is *whitebox cryptography*. It was proposed to prevent the extraction of a cryptographic key from the binary by mixing it with the algorithm. In the mid-2000s, the first implementations of whitebox algorithms for DES and AES were proposed [Chow et al. 2003a, 2003b; Link and Neumann 2005; Bringer et al. 2006]. However, all of them have been broken using techniques such as fault injection [Jacob et al. 2003], statistical analysis [Link et al. 2004], condensed implementation [Wyseur and Preneel 2005], differential cryptanalysis [Goubin et al. 2007; Wyseur et al. 2007; Billet et al. 2005; De Mulder et al. 2010], or generic cryptanalysis [Michiels et al. 2009]. Given this mixed history, in recent years, research on whitebox cryptography focused on the question how the general idea and its security concepts can be backed by a theoretical foundation. Wyseur [2009] discussed the state of the art of *whitebox cryptography* and proposed new block ciphers and design principles for the construction of whitebox cryptographic algorithms. Saxena et al. [2009] described a theoretical model of *whitebox cryptography* using appropriate security notions and presented both positive and negative results on whitebox cryptography. This leads us to the conclusion that in its current state the strength of *whitebox cryptography* is unproven, although it can make the extraction of the cryptographic key considerably more complex. Thus, we marked the obfuscation class *converting static data to procedures* in gray. Other obfuscation techniques that can provide at least limited resilience in this analysis scenario are *environmental requirements* and *hardware-assisted code obfuscation* because of their dependencies on external factors such as the presence of a particular hardware token. The strength of dongles for software protection was evaluated by Piazzalunga et al. [2007]. The authors developed a model for forecasting the amount of time an analyst would need to break

dongle-based software protection schemes and concluded that today's available dongle solutions provide only minimal protection. Still, we marked the class of *hardware-assisted code obfuscation* techniques in gray as dongles are only one simple concept in this class of obfuscations. Several approaches that are more resilient to dynamic analysis were introduced in the literature [Fu et al. 2007; Zhuang et al. 2004; Bitansky et al. 2011].

Locating Code. Some obfuscation techniques break abstractions in the code that aid human understanding. *Name scrambling*, *removing standard library calls*, and *breaking relations* do not prevent an automated tool from analyzing a program. However, they can make manual analysis by a human more difficult.

Other classes of code obfuscation have limited robustness against a human analyst trying to locate code. For example, Rolles [2009] introduced a semiautomated de-obfuscation approach against virtualization-obfuscation, which is based on reverse engineering the virtual machine, extracting the byte code, and then turning it into native code. Madou et al. [2006b, 2006c] developed an interactive deobfuscation tool named Loco that allows an analyst to navigate through the control flow graph of a program to undo static obfuscating transformations such as *control flow flattening*. Quist and Liebrock [2009] demonstrated how a sophisticated visual representation of the control flow of a program can speed up the analysis process. In particular, unpacking routines of malware can be identified efficiently using a visualization approach.

Extracting Code. Finding the location of code is a prerequisite for code extraction. Thus, similarly to the scenario of dynamic analysis, the extraction of code can be considered at least as difficult as locating code for a human analyst. One of the most popular approaches for human-assisted code extraction is *program slicing*, which is based on the idea of reducing the program code to a minimum *slice* that still produces a particular behavior or affects the value of a particular variable. In the literature, a multitude of program slicing approaches have been introduced over the past two decades (e.g., Lanubile and Visaggio [1997] and Ning et al. [1993]). A major limitation of program slicing is that the human analyst needs a deep understanding of the program internals to be able to specify a slicing criterion such as relevant variables and behavior. Several attempts have been made to raise the level of abstraction in slicing and thus make it less depended on manual analysis, such as conditioned slicing [Canfora et al. 1994, 1998; Cimitile et al. 1996; Danicic et al. 2004] and constraint slicing [Fox et al. 2004; Danicic et al. 2005; Field et al. 1995]. We marked *opaque predicates* and *aliasing* in gray as these obfuscation techniques can make the identification of the minimal subset that still implements a particular functionality more difficult.

Understanding Code. Despite the fact that getting a comprehensive understanding of the program structure and functionality can be considered the most ambitious aim of a human analyst, today's state of the art in code obfuscation provides only limited protection in this analysis scenario. This assumption is backed by a plethora of reports about successfully removed copy protection schemes for digital media such as CSS (DVD copy protection), Windows Media DRM [Myska 2009], and high-bandwidth digital content protection (HDCP) [Lomb and Guneysu 2011]. Another piece of evidence for this assumption is the successful reverse engineering of the VoIP (Voice over IP) software Skype. While Skype is known for its extensive use of code obfuscation, Biondi and Desclaux [2006] still were able to reveal the internal structure of the software. Furthermore, the client software of the cloud service provider Dropbox was successfully reverse engineered despite being heavily obfuscated [Kholia and Wegrzyn 2013]. A decisive factor for these recent success stories in code analysis are today's sophisticated reverse engineering tools that have become better and better in dealing

with obfuscated code [Eagle 2008; Ferguson and Kaminsky 2008; Eilam 2005]. For instance, the F.L.I.R.T. library of IDA Pro enables the recognition of standard library functions generated by a variety of different compilers. It can be concluded that almost all code obfuscation techniques have to be considered ineffective against a human analyst that puts enough time and effort into manual deobfuscation. We marked *name scrambling*, *removing standard library calls*, *breaking relations*, and *virtualization* in gray as these obfuscations can make manual analysis by a human more difficult. Furthermore, the obfuscation classes *environmental requirements* and *hardware-assisted code obfuscation* can be considered as strong against human analysis as long as the external requirement cannot be accessed by the analyst. *Antidebugging and Antidisassembly* was also marked in gray as all human-assisted analysis approaches described in the literature are still based on automated static and dynamic analysis tools.

6. CONCLUSIONS

In this survey, we addressed the question regarding to what extent software obfuscation is able to provide reasonable protection of programs against state-of-the-art code analysis techniques and tools. Despite more than two decades of research on obfuscation theory, reliable concepts for the evaluation of the resilience of an obfuscation technique have still not been found, and there are similar limitations for the evaluation of code analysis and deobfuscation techniques. With these constraints in mind, we conducted a literature review of code analysis techniques applied against different classes of obfuscations in specific attack scenarios. Where we could not find direct evidence of the resilience of an obfuscation in a particular analysis scenario in the literature, we made and justified our own inferences. Hence, this survey should not be seen as a formal analysis but as a snapshot of the current state in the arms race between obfuscation and code analysis. We considered the effects of obfuscations applied in isolation of each other, but we would like to point out that the analysis of combinations of multiple types of obfuscations constitutes an interesting area for future research.

Our results indicate that the strength of obfuscations heavily depends on the goals of the analyst and the available resources. Most of the more heavyweight analysis approaches introduced in academia have only been demonstrated to work on relatively small and specific examples, whereas large real-world programs can be significantly harder to analyze. A major limiting factor for code analysis is that the high complexity of the analysis problems often exceeds the resources available to the analyst. Therefore, simple obfuscations can still be quite effective where the deployed techniques have to be fast and lightweight, like many pattern matching or static analysis algorithms. This explains the unbroken popularity of software obfuscation among malware writers. Where more resources are available to run an expensive dynamic analysis or even perform manual reverse engineering, obfuscations are much less effective. As a result, intellectual property protection against a human adversary remains challenging.

Another observation is that much of the current academic research on code analysis in the presence of obfuscations focuses on malware. Frequently, the relevant literature describes methods for classifying programs as malicious based on identifying obfuscation techniques that are common in malware. The analysis of the actual functionality of an obfuscated program is typically left out of scope. Substantially less academic research has been done on reverse engineering obfuscated general (nonmalicious) programs, and tool support is worse.

The arms race between software obfuscation and analysis is still ongoing and the fundamental challenge of devising software protection mechanisms that are resistant against a human analyst remains. With today's software obfuscation techniques one has to assume that a dedicated analyst who is willing to spend enough time and effort will always be able to successfully analyze a program. Nevertheless, as this survey shows,

specific classes of software obfuscation can be effective in more restricted analysis scenarios.

REFERENCES

Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.

B. Anckaert, B. De Sutter, and K. De Bosschere. 2004. Software piracy prevention through diversity. In *Proceedings of the 4th ACM Workshop on Digital Rights Management*. ACM, New York, NY, 63–71.

B. Anckaert, M. Jakubowski, and R. Venkatesan. 2006. Proteus: Virtualization for diversified tamper-resistance. In *Proceedings of the ACM Workshop on Digital Rights Management*. ACM, New York, NY, 47–58.

Bertrand Anckaert, Mariusz H. Jakubowski, Ramarathnam Venkatesan, and Chit Wei Saw. 2009. Runtime protection via dataflow flattening. In *Proceedings of the 3rd International Conference on Emerging Security Information, Systems and Technologies (SECURWARE'09)*. IEEE, 242–248.

B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. 2007. Program obfuscation: A quantitative approach. In *Proceedings of the 2007 ACM Workshop on Quality of Protection*. ACM, New York, NY, 15–20.

G. Avoine, P. Junod, and P. Oechslin. 2007. *Computer System Security: Basic Concepts and Solved Exercises*. EPFL Press.

D. F. Bacon, S. L. Graham, and O. J. Sharp. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv.* 26, 4 (1994), 345–420.

Gogul Balakrishnan and Thomas W. Reps. 2004. Analyzing memory accesses in x86 executables. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Vol. 2985. Springer, Berlin, 5–23.

Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. 2014. Protecting obfuscation against algebraic attacks. In *Advances in Cryptology–EUROCRYPT 2014*. Springer, Berlin, 221–238.

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

Sébastien Bardin, Philippe Herrmann, and Franck Védrine. 2011. Refinement-based CFG reconstruction from unstructured programs. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*. 54–69.

U. Bayer, C. Kruegel, and E. Kirda. 2006. TTAnalyze: A tool for analyzing malware. In *Proceedings of the 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR'06)*.

Daniel Bilar. 2007. Opcodes as predictor for malware. *Int. J. Electron. Security Digital Forens.* 1, 2 (2007), 156–168.

Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. 2005. Cryptanalysis of a white box AES implementation. In *Proceedings of the 11th International Conference on Selected Areas in Cryptography*. Springer, Berlin, 227–240.

Philippe Biondi and Fabrice Desclaux. 2006. Silver needle in the skype. *Black Hat Eur.* 6 (2006), 25–47.

Nir Bitansky, Ran Canetti, Henry Cohn, Shafi Goldwasser, Yael Tauman Kalai, Omer Paneth, and Alon Rosen. 2014. The impossibility of obfuscation with auxiliary input or a universal simulator. In *Advances in Cryptology–CRYPTO 2014*. Springer, Berlin, 71–89.

Nir Bitansky, Ran Canetti, Shafi Goldwasser, Shai Halevi, Yael Tauman Kalai, and Guy N. Rothblum. 2011. Program obfuscation with leaky hardware. In *Advances in Cryptology–Asiacrypt 2011*. Vol. 7073. Springer, Berlin, 722–739.

Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, New York, NY.

Zvika Brakerski and Guy N. Rothblum. 2014. Virtual black-box obfuscation for all circuits via generic graded encoding. In *Theory of Cryptography*. Springer, Berlin, 1–25.

Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. In *Blackhat 2012*.

Murray Brand. 2010. *Analysis Avoidance Techniques of Malicious Software*. Ph.D. Dissertation. Edith Cowan University.

Julien Bringer, Herve Chabanne, and Emmanuelle Dottax. 2006. White box cryptography: Another attempt. *IACR Cryptology Eprint Archive* 2006 (2006).

Tom Brosch and Maik Morgenstern. 2006. Runtime packers: The hidden problem. *Black Hat USA*. Retrieved from https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A binary analysis platform. In *Proceedings of the 23th International Conference on Computer Aided Verification (CAV'11)*. 463–469.

David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP'08)*. IEEE, 143–157.

D. Bruschi, L. Martignoni, and M. Monga. 2006a. Detecting self-mutating malware using control-flow graph matching. *Detection of Intrusions and Malware & Vulnerability Assessment* (2006), 129–143.

Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006b. Using code normalization for fighting self-mutating malware. In *Proceedings of the International Symposium on Secure Software Engineering*. 37–44.

Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. 2010. Binary code extraction and interface identification for security applications. In *Proceedings of Network and Distributed System Security Symposium (NDSS'09)*.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*. 322–335.

Joan Calvet, José M. Fernandez, and Jean-Yves Marion. 2012. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 169–182.

R. Canetti and R. Dakdouk. 2008. Obfuscating point functions with multibit output. *Advances in Cryptology–Eurocrypt 2008* (2008), 489–508.

Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. 1998. Conditioned program slicing. *Inform. Software Technol.* 40, 11 (1998), 595–607.

Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. 1994. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance (ICSM'94)*. IEEE, 424–433.

Jan Cappaert, Nessim Kisserli, Dries Schellekens, and Bart Preneel. 2006. Self-encrypting code to protect against analysis and tampering. In *Proceedings of the 1st Benelux Workshop on Information and System Security*.

Jan Cappaert and Bart Preneel. 2010. A general model for hiding control flow. In *Proceedings of the 10th Annual ACM Workshop on Digital Rights Management*. ACM, New York, NY, 35–42.

Hoi Chang and Mikhail J. Atallah. 2002. Protecting software code by guards. In *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*. Springer, Berlin, 160–175.

Mohamed R. Chouchane and Arun Lakhotia. 2006. Using engine signature to detect metamorphic malware. In *Proceedings of the 4th ACM Workshop on Recurring Malcode*. ACM, New York, NY, 73–78.

S. Chow, P. Eisen, H. Johnson, and P. Van Oorschot. 2003a. White-box cryptography and an AES implementation. In *Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*. Springer, Berlin, 250–270.

Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. Van Oorschot. 2003b. A white-box DES implementation for DRM applications. In *Digital Rights Management*. Vol. 2696. Springer, Berlin, 1–15.

Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. 2001. An approach to the obfuscation of control-flow of sequential computer programs. In *Information Security*. Springer, Berlin, 144–155.

Mihai Christodorescu, Somesh Jha, Johannes Kinder, Stefan Katzenbeisser, and Helmut Veith. 2007. Software transformations to improve malware detection. *J. Comput. Virol.* 3, 4 (2007), 253–265.

M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. 2005. Semantics-aware malware detection. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*. IEEE, 32–46.

Cristina Cifuentes and K. John Gough. 1995. Decompilation of binary programs. *Software Pract. Exp.* 25, 7 (1995), 811–829.

Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. 1996. A specification driven slicing process for identifying reusable functions. *J. Software Maint. Res. Pract.* 8, 3 (1996), 145–178.

F. B. Cohen. 1993. Operating system protection through program evolution. *Comput. Security* 12, 6 (1993), 565–584.

Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional.

C. Collberg, C. Thomborson, and D. Low. 1997. *A Taxonomy of Obfuscating Transformations*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.

Christian Collberg, Clark Thomborson, and Douglas Low. 1998a. Breaking abstractions and unstructuring data structures. In *Proceedings of the 1998 International Conference on Computer Languages*. IEEE, 28–38.

C. Collberg, C. Thomborson, and D. Low. 1998b. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 184–196.

Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. 2010. Identifying dormant functionality in malware programs. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. IEEE, 61–76.

Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. 2009. Automatic static unpacking of malware binaries. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*. IEEE, 167–176.

K. Coogan, G. Lu, and S. Debray. 2011. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 275–284.

Bart Coppens, Bjorn De Sutter, and Jonas Maebe. 2013. Feedback-driven binary code diversification. *ACM Trans. Arch. Code Optimiz. (TACO)* 9, 4 (2013).

Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. 2008. Digging for data structures. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI'08)*.

Jedidiah R. Crandall, Gary Wassermann, Daniela A. S. de Oliveira, Zhendong Su, S. Felix Wu, and Frederic T. Chong. 2006. Temporal search: Detecting hidden malware timebombs with virtual machines. *ACM SIGPLAN Not.* 41, 11 (2006), 25–36.

Mila Dalla Preda and Roberto Giacobazzi. 2005. Semantic-based code obfuscation by abstract interpretation. In *Automata, Languages and Programming*. Springer, Berlin, 1325–1336.

M. Dalla Preda, R. Giacobazzi, S. Debray, K. Coogan, and G. Townsend. 2011. Modelling metamorphism by abstract interpretation. In *Proceedings of the 17th Annual Symposium onStatic Analysis*. 218–235.

M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. 2006. Opaque predicates detection by abstract interpretation. *Algebr. Methodol. Software Technol.* (2006), 81–95.

Sebastian Danicic, Mohammed Daoudi, Chris Fox, Mark Harman, Robert M. Hierons, John R. Howroyd, Lahcen Ourabya, and Martin Ward. 2005. Consus: A light-weight program conditioner. *J. Syst. Software* 77, 3 (2005), 241–262.

Sebastian Danicic, Andrea De Lucia, and Mark Harman. 2004. Building executable union slices using conditioned slicing. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*. IEEE, 89–97.

Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. New York, NY, 57–68.

Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. XIFER: A software diversity tool against code-reuse attacks. In *Proceedings of the 4th ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3'12)*.

Yoni De Mulder, Brecht Wyseur, and Bart Preneel. 2010. Cryptanalysis of a perturbated white-box AES implementation. In *Progress in Cryptology—INDOCRYPT 2010*. Springer, Berlin, 292–310.

B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. De Bosschere. 2009. Instruction set limitation in support of software diversity. *Inform. Security Cryptol.* (2009), 152–165.

Saumya Debray and Jay Patel. 2010. Reverse engineering self-modifying code: Unpacker extraction. In *17th Working Conference on Reverse Engineering (WCRE'10)*. IEEE, 131–140.

N. Dedić, M. Jakubowski, and R. Venkatesan. 2007. A graph game model for software tamper protection. In *Proceedings of the 9th International Conference on Information Hiding*. Springer-Verlag, 80–95.

J. C. Deprez and A. Lakhotia. 2000. A formalism to automate mapping from program features to code. In *Proceedings of the 8th International Workshop on Program Comprehension*. IEEE, 69–78.

Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. 2013. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. ACM, New York, NY, 61–72.

Chris Eagle. 2008. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press.

Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.* 44, 2 (2012).

Eldad Eilam. 2005. *Reversing: Secrets of Reverse Engineering*. Wiley, New York, NY.

M. V. Emmerik and Trent Waddington. 2004. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering*. IEEE, 27–36.

Justin Ferguson and Daniel Kaminsky. 2008. *Reverse Engineering Code with IDA Pro*. Syngress.

John Field, Ganesan Ramalingam, and Frank Tip. 1995. Parametric program slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 379–392.

Halvar Flake. 2004. Structural comparison of executable objects. In *Proceedings of the Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop (DIMVA'04)*. 161–173.

Christophe Foket, Bjorn De Sutter, Bart Coppens, and Koen De Bosschere. 2013. A novel obfuscation: Class hierarchy flattening. In *Foundations and Practice of Security*. Springer, Berlin, 194–210.

Christophe Foket, Bjorn De Sutter, and Koen De Bosschere. 2014. Pushing java type obfuscation to the limit. *IEEE Trans. Dependable Secure Comput.* 6 (2014), 553–567.

Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*. IEEE, 67–72.

Chris Fox, Sebastian Danicic, Mark Harman, and Robert M. Hierons. 2004. ConSIT: A fully automated conditioned program slicer. *Software: Pract. Exp.* 34, 1 (2004), 15–46.

Michael Franz. 2010. E. unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*. ACM, New York, NY, 7–16.

Bin Fu, Sai Aravalli, and John Abraham. 2007. Software protection by hardware and obfuscation. In *Proceedings of the 2007 International Conference on Security & Management (SAM'07)*. 367–373.

Debin Gao, Michael K. Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*. Springer, Berlin, 238–255.

Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. 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'13)*. IEEE, 40–49.

Sudeep Ghosh, Jason D. Hiser, and Jack W. Davidson. 2010. A secure and robust approach to software tamper resistance. In *Information Hiding*. Springer, Berlin, 33–47.

Roberto Giacobazzi. 2008. Hiding information in completeness holes: New perspectives in code obfuscation and watermarking. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM'08)*. IEEE, 7–18.

Roberto Giacobazzi and Isabella Mastroeni. 2012. Making abstract interpretation incomplete: Modeling the potency of obfuscation. In *Proceedings of the 19th International Symposium Static Analysis (SAS'12)*. Springer, Berlin, 129–145.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. 213–223.

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated whitebox fuzz testing. In *Proceedings of Network and Distributed System Security Symposium (NDSS'08)*.

Shafi Goldwasser and Guy N. Rothblum. 2007. On best-possible obfuscation. In *Theory of Cryptography*. Vol. 4392. Springer, Berlin, 194–213.

L. Goubin, J. M. Masereel, and M. Quisquater. 2007. Cryptanalysis of white box DES implementations. In *Selected Areas in Cryptography*. Vol. 4876. Springer, Berlin, 278–295.

K. Griffin, S. Schneider, X. Hu, and T. Chiueh. 2009. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection*. Lecture Notes in Computer Science, Vol. 5758. Springer, Berlin, 101–120.

Felix Gröbert, Carsten Willems, and Thorsten Holz. 2011. Automated identification of cryptographic primitives in binary programs. In *Recent Advances in Intrusion Detection*. Lecture Notes in Computer Science, Vol. 6961. Springer, Berlin, 41–60.

Derrick Grover. 1992. *Protection of Computer Software: Its Technology and Application*. Cambridge University Press, Cambridge.

Y. Guillot and A. Gazet. 2010. Automatic binary deobfuscation. *J. Comput. Virol.* 6, 3 (2010), 261–276.

Wadie Guizani, J.-Y. Marion, and Daniel Reynaud-Plantey. 2009. Server-side dynamic code analysis. In *Proceedings of the 2009 4th International Conference on Malicious and Unwanted Software (MALWARE'09)*. IEEE, 55–62.

L. C. Harris and B. P. Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH Comput. Arch. News* 33, 5 (2005), 63–68.

Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E. Tarjan. 2002. Dynamic self-checking techniques for improved tamper resistance. In *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*. Springer, Berlin, 141–159.

S. Horwitz. 1997. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.* 19, 1 (1997), 1–6.

Grégoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel, and Giovanni Vigna. 2012. A static, packer-agnostic filter to detect similar malware samples. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer-Verlag, 102–122.

M. Jacob, D. Boneh, and E. Felten. 2003. Attacking an obfuscated cipher by injecting faults. *Digital Rights Manag.* (2003), 16–31.

Matthias Jacob, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. 2007. Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *Proceedings of the 9th Workshop on Multimedia & Security*. ACM, New York, NY, 129–140.

M. Jakubowski, P. Naldurg, V. Patankar, and R. Venkatesan. 2007. Software integrity checking expressions (ICEs) for robust tamper detection. In *Information Hiding*. Vol. 4567. Springer, Berlin, 96–111.

Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode*. ACM, New York, NY, 46–53.

Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. 2003. Exploiting self-modification mechanism for program protection. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*. IEEE, 170–179.

Abhishek Karnik, Suchandra Goswami, and Ratan Guha. 2007. Detecting obfuscated viruses using cosine similarity analysis. In *Proceedings of the 1st Asia International Conference on Modelling & Simulation (AMS'07)*. IEEE, 165–170.

Dhiru Kholia and Przemysław Wegrzyn. 2013. Looking inside the (drop)box. In *Proceedings of the 7th Usenix Workshop on Offensive Technologies (Woot'13)*.

Johannes Kinder. 2012. Towards static analysis of virtualization-obfuscated binaries. In *Proceedings of the 19th Working Conference Reverse Engineering (WCRE 2012)*. IEEE, 61–70.

Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. 2005. Detecting malicious code by model checking. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Vol. 3548. Springer, Berlin, 174–187.

J. Kinder and H. Veith. 2008. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*. Springer, Berlin, 423–427.

J. Kinder, F. Zuleger, and H. Veith. 2009. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09)*. Springer, Berlin, 214–228.

James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

S. T. King and P. M. Chen. 2006. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*. IEEE.

C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. 2010. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. IEEE, 29–44.

Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. 2011. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 285–296.

Christopher Krügel, William K. Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *Proceedings of the USENIX Security Symposium*. 255–270.

Arun Lakhotia, Davidson R. Boccardo, Anshuman Singh, and Aleardo Manacero Jr. 2010. Context-sensitive analysis without calling-context. *Higher-Order Symbol. Comput.* 23, 3 (2010), 275–313.

Filippo Lanubile and Giuseppe Visaggio. 1997. Extracting reusable functions by flow graph based program slicing. *IEEE Trans. Software Eng.* 23, 4 (1997), 246–259.

Tımea László and Ákos Kiss. 2009. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis De Rolando Eötvös Nominatae, Sectio Computatorica* 30 (2009), 3–19.

Felix Leder, Peter Martini, and Andre Wichmann. 2009. Finding and extracting crypto routines from malware. In *IEEE 28th International Performance Computing and Communications Conference (IPCCC'09)*. IEEE, Washington, DC, 394–401.

J. Li, M. Xu, N. Zheng, and J. Xu. 2009. Malware obfuscation detection via maximal patterns. In *Proceedings of the 3rd International Symposium on Intelligent Information Technology Application (LITA'09)*, Vol. 2. IEEE, 324–328.

Z. Lin, X. Zhang, and D. Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Network and Distributed System Security Symposium*.

Hamilton E. Link and William D. Neumann. 2005. Clarifying obfuscation: Improving the security of white-box des. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, Vol. 1. IEEE, 679–684.

Hamilton E. Link, Richard Crabtree Schroeppel, William Douglas Neumann, Philip LaRoche Campbell, Cheryl Lynn Beaver, Lyndon George Pierson, and William Erik Anderson. 2004. *Securing Mobile Code*. Technical Report. Sandia National Laboratories.

C. Linn and S. Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 290–299.

Benno Lomb and Tim Guneysu. 2011. Decrypting HDCP-protected video streams using reconfigurable hardware. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig'11)*. IEEE, 249–254.

B. Lynn, M. Prabhakaran, and A. Sahai. 2004. Positive results and techniques for obfuscation. In *Advances in Cryptology–Eurocrypt 2004*. Springer, Berlin, 20–39.

Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere, Jan Cappaert, and Bart Preneel. 2006. On the effectiveness of source code transformations for binary obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'06)*. 527–533.

Matias Madou, Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. 2005. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM Workshop on Digital Rights Management*. ACM, New York, NY, 75–82.

Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. 2006a. Software protection through dynamic code mutation. In *Information Security Applications*. Springer, Berlin, 194–206.

M. Madou, L. Van Put, and K. De Bosschere. 2006b. LOCO: An interactive code (de) obfuscation tool. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, New York, NY, 140–144.

M. Madou, L. Van Put, and K. De Bosschere. 2006c. Understanding obfuscated code. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 268–274.

A. Majumdar, A. Monsifrot, and C. Thomborson. 2006. On evaluating obfuscatory strength of alias-based transforms using static analysis. In *Proceedings of the International Conference on Advanced Computing and Communications (ADCOM'06)*. IEEE, 605–610.

Anirban Majumdar and Clark Thomborson. 2006. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*. Australian Computer Society, 187–196.

Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. 2009. English shellcode. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 524–533.

Aleksandr Matrosov, Eugene Rodionov, David Harley, and Juraj Malcho. 2010. Stuxnet under the microscope. *ESET LLC (September 2010)* (2010).

Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. 2011. A taxonomy of self-modifying code for obfuscation. *Comput. Security* 30, 8 (2011), 679–691.

Wil Michiels, Paul Gorissen, and Henk D. L. Hollmann. 2009. Cryptanalysis of a generic class of white-box implementations. In *Selected Areas in Cryptography*. Vol. 5381. Springer, Berlin, 414–428.

Craig Miles, Arun Lakhotia, and Andrew Walenstein. 2012. In situ reuse of logically extracted functional components. *J. Comput. Virol.* 8, 3 (2012), 73–84.

Akito Monden, Antoine Monsifrot, and Clark Thomborson. 2004. A framework for obfuscated interpretation. In *Proceedings of the 2nd Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation-Volume 32*. Australian Computer Society, 7–16.

Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007a. Exploring multiple execution paths for malware analysis. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*. IEEE, 231–245.

A. Moser, C. Kruegel, and E. Kirda. 2007b. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*. IEEE, 421–430.

M. Myska. 2009. The true story of DRM. *Masaryk Ujl & Tech.* 3 (2009), 267–278.

C. Nachenberg. 1997. Computer virus-coevolution. *Commun. ACM* 50, 1 (1997), 46–51.

Vijayanand Nagarajan, Rajiv Gupta, Xiangyu Zhang, Matias Madou, and Bjorn De Sutter. 2007. Matching control flow of program versions. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'07)*. IEEE, 84–93.

J. Newsome, B. Karp, and D. Song. 2005. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*. IEEE, 226–241.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer, Berlin.

Jim Q Ning, Andre Engberts, and Wojtek Kozaczynski. 1993. Recovering reusable components from legacy systems by program segmentation. In *Proceedings of the Working Conference on Reverse Engineering*. IEEE, 64–72.

Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. 2000. Experience with software watermarking. In *Proceedings of the 16th Annual Conference on Computer Security Applications (ACSAC'00)*. IEEE, 308–316.

Ugo Piazzalunga, Paolo Salvaneschi, Francesco Balducci, Pablo Jacomuzzi, and Cristiano Moroncelli. 2007. Security strength measurement for dongle-protected software. *IEEE Security Privacy* 5, 6 (2007), 32–40.

Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. 2007. Binary obfuscation using signals. In *Proceedings of the Usenix Security Symposium*. 275–290.

Daniel A. Quist and Lorie M. Liebrock. 2009. Visualizing compiled executables for malware analysis. In *Proceedings of the 6th International Workshop on Visualization for Cyber Security, 2009 (VizSec'09)*. IEEE, 27–32.

Jason Raber and Eric Laspe. 2007. Deobfuscator: An automated approach to the identification and removal of code obfuscation. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07)*. IEEE, 275–276.

G. Ramalingam. 1994. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1467–1471.

J. Riordan and B. Schneier. 1998. Environmental key generation towards clueless agents. *Mobile Agents and Security* (1998), 15–24.

R. Rolles. 2009. Unpacking virtualization obfuscators. In *Proceedings of the 3rd Usenix Workshop on Offensive Technologies (Woot'09)*.

Kevin A. Roundy and Barton P. Miller. 2013. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.* 46, 1 (2013).

P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. 2006. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 289–300.

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

Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. 2005. Java obfuscation approaches to construct tamper-resistant object-oriented programs. *IPSJ Digital Courier* 1 (2005), 349–361.

Amitabh Saxena, Brecht Wyseur, and Bart Preneel. 2009. Towards security notions for white-box cryptography. In *Information Security*. Springer, Berlin, 49–58.

S. Schrittwieser and S. Katzenbeisser. 2011. Code obfuscation against static and dynamic reverse engineering. In *Proceedings of the 13th International Conference on Information Hiding (IH'11)*. Springer, Berlin, 270–284.

Sebastian Schrittwieser, Stefan Katzenbeisser, Peter Kieseberg, Markus Huber, Manuel Leithner, Martin Mulazzani, and Edgar Weippl. 2013. Covert computation: Hiding code in code for obfuscation purposes. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ACM, 529–534.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010*. 317–331.

Edward J. Schwartz, J. Lee, Maverick Woo, and David Brumley. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the Usenix Security Symposium*.

Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of executable code revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE, 45–54.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 263–272.

Hovav Shacham. 2007. 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*. ACM, New York NY, 552–561.

Adi Shamir and Nicko Van Someren. 1999. Playing'hide and seek' with stored keys. In *Financial Cryptography*. Vol. 1648. Springer, Berlin, 118–124.

M. Sharif, A. Lanzi, J. Giffin, and W. Lee. 2009. Automatic reverse engineering of malware emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. IEEE, 94–109.

M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee. 2008. Eureka: A framework for enabling static malware analysis. *Computer Security-Esorics 2008* (2008), 481–500.

Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'08)*.

A. Slowinska, T. Stancescu, and H. Bos. 2011. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)*.

Harry M. Sneed. 2000. Encapsulation of legacy software: A technique for reusing legacy software components. *Ann. Software Eng.* 9, 1–2 (2000), 293–313.

Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote Invited Paper.*

Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. 2010. On the infeasibility of modeling polymorphic shellcode. *Mach. Learn.* 81, 2 (2010), 179–205.

Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon. 2003. Obfuscation of design intent in object-oriented applications. In *Proceedings of the 3rd ACM Workshop on Digital Rights Management*. ACM, New York, NY, 142–153.

Joe Stewart. 2006. Ollybone: Semi-automatic unpacking on IA-32. In *Proceedings of the 14th Def Con Hacking Conference*.

Y. Tang and S. Chen. 2007. An automated signature-based approach against polymorphic internet worms. *IEEE Trans. Parallel Distrib. Syst.* 18, 7 (2007).

A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. 2010. Directed proof generation for machine code. In *Proceedings of the 22th International Conference on Computer Aided Verification (CAV'10)*. Springer, Berlin, 288–305.

S. R. Tilley, S. Paul, and D. B. Smith. 1996. Towards a framework for program understanding. In *Proceedings of the 4th Workshop on Program Comprehension*. IEEE, 19–28.

S. Treadwell and M. Zhou. 2009. A heuristic approach for detection of obfuscated malware. In *Proceedings of the IEEE International Conference on Intelligence and Security Informatics (ISI'09)*. IEEE, 291–299.

H. Y. Tsai, Y. L. Huang, and D. Wagner. 2009. A graph approach to quantitative analysis of control-flow obfuscating transformations. *IEEE Trans. Inform. Forens. Security* 4, 2 (2009), 257–267.

S. K. Udupa, S. K. Debray, and M. Madou. 2005. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering*. IEEE.

Zeljko Vrba, Pål Halvorsen, and Carsten Griwodz. 2010. Program obfuscation by strong cryptography. In *Proceedings of the International Conference on Availability, Reliability, and Security (ARES'10)*. IEEE, 242–247.

A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. 2006. Normalizing metamorphic malware using term rewriting. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*. IEEE, 75–84.

C. Wang, J. Davidson, J. Hill, and J. Knight. 2001. Protection of software-based survivability mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*. IEEE, 193–202.

C. Wang, J. Hill, J. Knight, and J. Davidson. 2000. *Software Tamper Resistance: Obstructing Static Analysis of Programs*. Technical Report. CS-2000-12, University of Virginia.

M. Webster and G. Malcolm. 2009. Detection of metamorphic and virtualization-based malware using algebraic specification. *J. Comput. Virol.* 5, 3 (2009), 221–245.

H. Wee. 2005. On obfuscating point functions. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*. ACM, New York, NY, 523–532.

N. Wilde and M. C. Scully. 2006. Software reconnaissance: Mapping program features to code. *J. Software Maint.: Res. Pract.* 7, 1 (2006), 49–62.

Carsten Willems and Felix C. Freiling. 2012. Reverse code engineering-state of the art and countermeasures. *Inform. Technol.* 54, 2 (2012), 53–63.

M. J. Wolfe, C. Shanklin, and L. Ortega. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman, Reading, MA.

Z. Wu, S. Gianvecchio, M. Xie, and H. Wang. 2010. Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 536–546.

Brecht Wyseur. 2009. *White-Box Cryptography*. Ph.D. Dissertation. KU Leuven.

B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. 2007. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In *Proceedings of the 14th International Conference on Selected Areas in Cryptography*. Springer, Berlin, 264–277.

Brecht Wyseur and Bart Preneel. 2005. Condensed white-box implementations. In *Proceedings of the 26th Symposium on Information Theory in the Benelux*. 296–301.

Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *Proceedings of the 22nd Network and Distributed Systems Security Symposium (NDSS)*.

Heng Yin and Dawn Song. 2010. *TEMU: Binary Code Analysis Via Whole-System Layered Annotative Execution*. Technical Report UCB/EECS-2010-3. EECS Department, University of California, Berkeley.

Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2013. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*. ACM, New York, NY.

Xiangyu Zhang and Rajiv Gupta. 2005. Matching execution histories of program versions. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, New York, NY 197–206.

Z. Zhao, G. J. Ahn, and H. Hu. 2011. Automatic extraction of secrets from malware. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE'11)*. IEEE, 159–168.

Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. 2007. Information hiding in software with mixed boolean-arithmetic transforms. In *Information Security Applications*. Springer, Berlin, 61–75.

X. Zhuang, T. Zhang, H. H. S. Lee, and S. Pande. 2004. Hardware assisted control flow obfuscation for embedded processors. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 292–302.

Received November 2013; revised June 2015; accepted January 2016